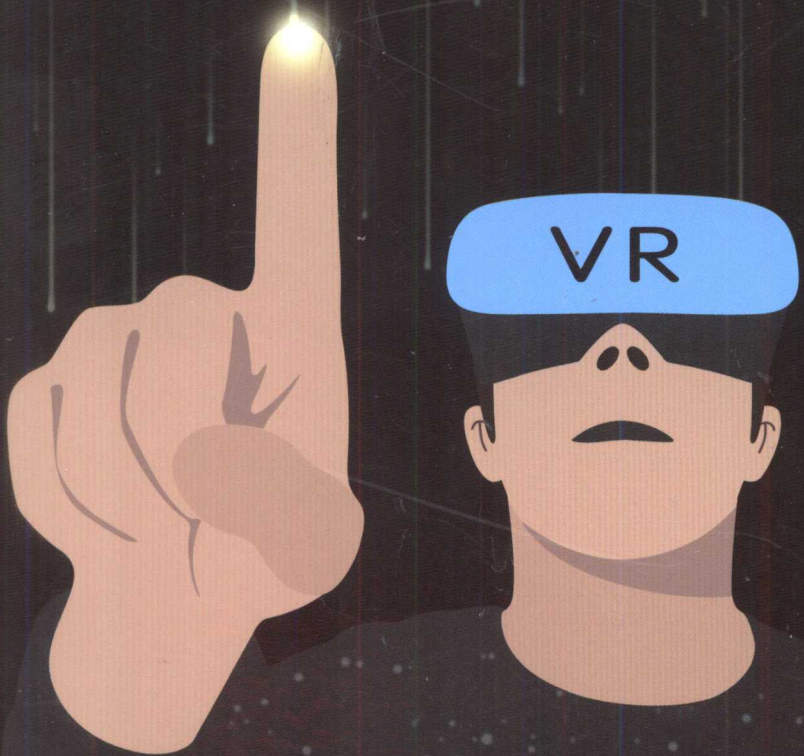
版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

VR三维技术系列

三维模型参数化算法：理论和实践（C#版本）

赵 辉 顾险峰 雷 娜 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VR 三维技术系列

三维模型参数化算法：理论和实践（C#版本）

赵 辉 顾险峰 雷 娜 著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

本书介绍了各种三维模型参数化算法。本书一共分为17章，详细讲述了曲面参数化、Blender软件贴图、碟形网格参数化算法、三维模型扭曲度量、和谐参数化算法、迭代参数化算法、基于角度平展参数化算法、位置重建算法、LSCM算法、DCP/DAP算法、频谱参数化算法、局部全局参数化算法、高斯曲率参数化算法、线性系统和线性系统函数库的使用。本书包含了所有线性参数化算法的理论和实现。通过本书的学习，可以掌握三维模型贴图的原理以及线性系统在三维算法里的应用。

本书不仅可以作为数字媒体技术专业的专业基础课教材，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目(CIP)数据

三维模型参数化算法：理论和实践：C#版本 / 赵辉，顾险峰，雷娜著. —北京：电子工业出版社，2017.7
(VR三维技术系列)
ISBN 978-7-121-31682-1

Ⅰ. ①三… Ⅱ. ①赵… ②顾… ③雷… Ⅲ. ①三维动画软件－程序设计 Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2017)第120542号

策划编辑：张　迪
责任编辑：底　波
印　　刷：中国电影出版社印刷厂
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编100036
开　　本：787×1092　1/16　印张：16.5　字数：422千字
版　　次：2017年7月第1版
印　　次：2017年7月第1次印刷
定　　价：79.00元

所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：(010)88254469，zhangdi@phei.com.cn。

序

2015 年以来，虚拟现实技术的应用在国际国内发展很快。教育、医疗、娱乐、影视、游戏、安全、交通等各行各业都对虚拟现实技术进行了大量应用。虚拟现实技术的基础和核心是三维计算机图形学，分为四大模块：建模、渲染、动画、交互。目前国内大量的虚拟现实应用都局限于在西方开发的虚拟现实引擎的技术上进行开发的上层应用。我们这套丛书着重底层核心技术的讲解，三维计算机图形学在知识结构上来说需要数学、物理、工程、计算机编程、艺术 5 个方面。设计建模、渲染等算法需要微分几何、线性代数、概率统计等数学知识的理解和掌握；动画模拟需要流体、刚体等物理知识的理解和掌握；把这些数学、物理理论变为程序需要极强的编码能力，也就是从理论到实践的工程能力；三维图形学的最终表现形式是视觉上可看得到的，因此也需要良好的艺术修养和审美。虚拟现实和它所依赖的三维计算机图形学特别适合锻炼并能够融会贯通学生的数学、物理、工程、编程和艺术能力。三维计算机图形学是一个跨学科的领域，三维图形学处理的是三维模型数据，学生在这个领域中学到的数学建模、工程等能力，也可以用到其他行业，如人工智能等，对其他行业的大数据进行分析和处理。

2008 年以来，全国各个高等院校纷纷在各自软件工程学科专业的基础上开设了数字媒体技术专业。数字媒体技术专业和计算机科学专业的区别是，前者主要是着重学习二维图像和三维图形相关的算法和应用开发，而后者还需要学习其他计算机科学相关的知识。由于开设和建立时间短，各学校的数字媒体技术专业的教学工作都还处在摸索阶段，也没有形成统一、成熟的教材体系。根据在数字媒体技术专业多年的教学实践经验，我们总结出本专业要以计算机三维图形学的理论和算法为基础，以三维应用开发为导向进行建设。

根据多年一线教学经验与反馈，以及当前的三维图形学研究成果，我们编写了本套丛书。本套丛书涵盖了三维图形学算法的 3 个方面：建模、动画和渲染。内容根据数字媒体技术专业的教学特点分散到 5 本 VR 三维技术系列图书中。通过本系列专业图书，再加上已有的、成熟的计算机基础编程教材，以及三维软件使用的教材，就可以完整地覆盖数字媒体技术专业的所有课程。

书里的代码采用 C#编程语言。C#编程语言是一种结合了 C ++ 和 Java 优点的编程语言。C#语言相对于其他编程语言来说比较容易学习和掌握，但是本套丛书里讲述的原理和算法不仅限于 C#语言。读者可以通过示例中的代码，采用自己熟悉的编程语言来进行编程。本套丛书包含了很多计算机图形学会议 Siggraph 论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现和分析。

前　言

虚拟现实（Virtual Reality）是一项最近非常热门、应用广泛的技术。该技术起源于计算机三维图形学。计算机中这个方向的研究涉及大量的数学、物理、工程、编程以及艺术，需要跨学科协作。除了计算机专业，近几年在软件工程、数字媒体技术等专业也大量开设了这方面的课程。虚拟现实中一个最重要的应用就是模型贴图，即对模型进行纹理贴图，从而使模型看起来更逼真。模型贴图技术依赖于三维模型的参数化算法，也就是把模型从三维空间展开到二维平面，然后才能贴上二维的图片，最后再把两者的信息映射回三维模型上。

三维模型参数化算法涉及微分几何、线性代数等数学知识，可以分为保角参数化、保面积参数化及混合参数化等。保角参数化保持映射前后的角度、面积不变，这样能够保持原始模型的结构。保面积参数化目标是保持局部每个三角形的面积不发生改变。针对这两个目标，设计了各种各样的算法，有线性的，有非线性的。

通常来说，对于一个封闭的三维模型需要切开再进行参数化。但基于计算共形几何的理论，我们也可以不切开，对一个模型进行全局参数化。对于保面积参数化，也可以用最优传输的理论来实现。

本书介绍了各种三维模型参数化算法。本书一共分为17章，详细讲述了曲面参数化、Blender软件贴图、碟形网格参数化算法、三维模型扭曲度量算法、和谐参数化算法、迭代参数化算法、基于角度平展参数化算法、位置重建算法、LSCM算法、DCP/DAP算法、频谱参数化算法、局部全局参数化算法、高斯曲率参数化算法、线性系统和线性系统函数库的使用。本书包含了所有线性参数化算法的理论和实现。通过本书的学习，可以掌握三维模型贴图的原理以及线性系统在三维算法里的应用。

本书不仅可以作为数字媒体技术专业的专业基础课教材，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。需要书中部分代码的读者，可发邮件向作者索取，邮箱地址：graphicsresearch@qq.com。

赵　辉

2017年5月于清华大学近春园

作者简介

赵辉，虚拟现实专家、清华大学丘成桐数学科学中心访问学者、哈佛大学访问学者。主要研究计算微分几何、拓扑、三维模型处理算法（三维模型简化、细分、分割、变形、光滑、参数化、向量场、四边形化等）、三维动画算法（骨骼动画、蒙皮算法）、渲染算法（非真实感渲染、实时渲染、基于物理渲染），以及三维技术在3D打印、虚拟现实、增强现实、三维游戏、手机游戏、影视特效等的应用。

顾险峰，师从国际著名微分几何大师丘成桐院士，现为纽约州立大学石溪分校计算机科学系和应用数学系终身教授，清华大学丘成桐数学科学中心客座教授，大连理工大学海天学者，首都师范大学数字几何和成像实验室主任等。2005年获得美国国家自然科学基金CAREER奖，2006年获得中国国家自然科学基金海外杰出青年学者奖，2013年第六届世界华人数学家大会晨兴应用数学金奖等。

顾险峰教授和丘成桐先生及其合作者共同创立了一门新兴的跨领域学科：计算共形几何。这门学科结合了现代几何和计算机科学，广泛应用于计算机图形学、计算机视觉、可视化、几何建模、网络和医学图像等领域。

雷娜，大连理工大学软件学院教授，博士生导师，北京市成像技术高精尖创新中心兼职研究员；中国工业与应用数学学会几何设计与计算专业委员会委员；中国数学会计算机数学专业委员会委员；美国数学会Mathematical Review评论员；清华大学数学科学中心访问教授；纽约州立大学石溪分校计算机系访问教授；德克萨斯大学奥斯汀分校计算工程与科学研究所Research Fellow；中科院数学与系统科学研究院访问学者。主要研究兴趣是应用现代微分几何和代数几何的理论与方法解决工程及医学领域的问题，聚焦于计算共形几何、计算拓扑、符号计算及其在计算机图形学、计算机视觉、几何建模和医学图像中的应用。

目　录

第1章 浅谈曲面参数化

1.1 算法和理论（Ⅰ）

2017年寒假期间，笔者老顾游历了很多城市，包括武汉、南京、合肥、金华、青岛、郑州、北京、大连，同时访问了许多大学，和很多领域的专家学者探讨了学术问题，进行了一些演讲，从基础数学至计算机科学，再到具体应用。青年学者们的求知热情非常令人感动。因为每次演讲都是蜻蜓点水，无法将内容讲解透彻，因此老顾计划撰写几篇博文，将相关主题进行深入探讨。

从历史的全局观点来看，工程技术的发展不停地对纯粹科学提出挑战，从而促进理论的发展，同时反过来理论的建立指导着技术实践，为技术的进一步发展指明了方向。历史就是这样循环往复、螺旋上升的。但每个人的生命历程相对短暂，终其一生，只能窥见历史螺旋中的一小段弧线。并且，实际情形非常复杂，历史洪流经常有回旋湍流、泥沙俱下，渺小的个人被裹挟其中，经常身不由己、迷失方向，在无可奈何中随波逐流、蹉跎岁月。

在过去的科研教学中，笔者老顾与纯粹数学家和计算机科学家都有长期合作，他们的知识结构不同，价值观念迥异，在很大程度上分别代表了科学和技术两种文化理念。工程技术人员非常注重算法工艺的开发，强调实用价值，目的在于创新当前技术，推动时代的发展。其研究方法侧重于经验的积累，唯象方法的总结；基础科学家则专注于深刻理论的建立，忽视短期的经济回报，强调长远的价值，推动整体文明的发展。其研究方法侧重抽象理论的建构，苛求严格性、永恒性。

笔者老顾经常遇到计算机科学背景的学生和基础数学背景的学生，他们不同的知识结构和职业训练给了他们非常不同的价值取向和审美品位。很多计算机科学的学生刻意磨炼动手能力，经常陶醉于自己实现的算法程序，追求能够创业的新颖想法，但对于算法的严格性、算法所依据的理论完备性，没有苛求；基础数学的学生注重磨炼抽象思维能力，对于深刻宏大的理论孜孜以求，对于各种数学结构及其内在联系非常敏感，并且能够产生深刻的审美体验，但对于如何将抽象的理论应用于实践有所忽视。笔者老顾在清华大学、大连理工大学和首都师范大学都教授过计算共形几何课程。有很多计算机科学系的学生非常热衷学习并实现各种算法，但比较抵触拓扑和几何理论；也有数学系的学生专注而热切地学习相应的理论，但对于算法部分嗤之以鼻。其实，这些学生都是缺乏人生阅历的年轻人，所处的相对单纯的人文环境塑造了相对狭窄的学术视野。同时，年轻人所面临的学位、就业压力，所在领域的价值取向，使得他们无法自由地向学术纵深探索。例如，有一位计算机图形学领域的博士生，他的博士专题是曲面参数化。他花费了巨大的精力研读了计算机图形学领域顶级期刊中

所有的相关文章，由此构建了他这一方面的知识结构，并且将个人的改进创造融合到时下最为时髦的方法之中，发表在计算机科学的期刊上，得到了学术界的认可，获取了博士学位。但是，他所发明的算法依然停留在经验性方法阶段，他没有真正深究其后的理论，一方面是因为其知识结构的不足，无法真正在理论方面有所突破，另一方面也由于工程界忽视理论的严密性，完全认可经验性的方法，同时更是为生活压力所迫，力求尽早拿到学位。

笔者老顾也经常遇到社会地位稳定的学者们，他们对于技术、理论的问题的态度更为成熟而深邃。很多功成名就的计算机科学家，他们最大的遗憾就是自己的发明创造被时代所抛弃。比如，计算机图形学领域的泰斗级学者 Tom Sederberg 教授，有一次告诉笔者老顾他的亲身经历。Sederberg 教授刚出道时，计算机的存储和计算能力都非常有限，曲面的主要表示方法是样条曲面（Spline Surface），图形学中曲面渲染（Rendering）的主要方法是光线跟踪法（Ray Tracing）。光线跟踪法需要计算一条射线和曲面的交点，射线的方程是 $\boldsymbol{r}(t)=\boldsymbol{e}+t\boldsymbol{d}$，这里$(\boldsymbol{e},\boldsymbol{d})$代表相机的光心和射线方向；样条曲面是一种参数表示，$\boldsymbol{r}(u,v)=(x(u,v),y(u,v),z(u,v))$，需要被转换成所谓的隐式表示 $f(x,y,z)=0$，从而和射线方程联立，求得交点。Sederberg 教授的成名作就是用代数几何中的结式（Resultant）方法将样条曲面隐式化（Implicification）。但是，随着时代的发展，计算机存储和计算能力的提高，在数字媒体、电影游戏领域曲面都是用三角网格来表示的，曲面渲染算法主要是用 OpenGL，即便用光线跟踪法，样条曲面也是用三角网格来逼近。Sederberg 教授的隐式化方法迅速被遗忘，并且一度被评为几何建模领域十大无用算法之首。这件事情曾经深深地刺激了 Sederberg 教授，使得他发愤图强，发明了现在被广泛应用的 T－样条理论和算法。也有一些计算机科学家在某个领域独领风骚，发明了影响深远的算法，但是苦恼于无法深刻理解所研究现象的本质和其算法的适用范围。更多的计算机科学家也忧虑自己的发明具有鲜明的时效性，无法长久传世。纯粹数学家相对较少怀疑自身研究成果的恒久价值，但苦恼于无法被普罗大众所理解，距离国计民生过于遥远。

在大连理工大学演讲时，罗钟铉教授大发感慨："中国现在是大国，但还不是强国。中国有着数千年的历史，发明过璀璨的技术，但没有发展出完备成熟的理论体系。因此许多技术在近代失传。目前的社会比较急功近利，在喧嚣浮躁中更加需要追求深刻追求永恒的精神。"

如果我们将眼光放长远，考察科技发展历史，我们的确能够看到技术和理论相辅相成、相互促进的发展历程。在计算机图形学中有一个非常独特的领域：曲面参数化，其 20 年的发展历程佐证了技术和理论相互作用的历史发展观。我们计划首先介绍如何用工程方法来研究曲面参数化问题，然后介绍从古典几何理论如何理解这一问题，最后介绍如何基于古典理论来建立现代离散理论。

1.2 算法和理论（Ⅱ）

1. 时代的要求

20 世纪 90 年代，计算机图形学技术异军突起，特别是 GPU 技术的发明和普及，使得 CG 电影工业和电子游戏产业蓬勃发展。在 GPU 中，曲面数据处理被分成两条流水线：一条处理曲面的几何数据和几何变换，如曲面的平移旋转、射影变换等；另一条处理曲面的纹理

信息，如颜色、法向量场、局部几何细节鳞片结构、局部材质特性 BRDF 等。几何数据结构一般是多面体三角网格，纹理数据结构一般是平面图像。在图形渲染过程中，两条流水线的处理结果相互融合，将平面的二维纹理粘贴到弯曲的三维曲面上，这一技术称为**曲面纹理贴图**。

纹理贴图技术在数学上等价于求从曲面到平面区域的一个光滑双射，这称为所谓的**曲面参数化**问题。如图 1-1 所示，我们将各种曲面映射到平面区域，然后在平面参数区域上贴上纹理图像，通过逆映射将纹理图像拉回三维曲面。如图 1-2 所示，我们将大理石纹理贴在米开朗基罗的大卫头像上面，从而获得了大理石雕像的质感。

图 1-1　曲面参数化

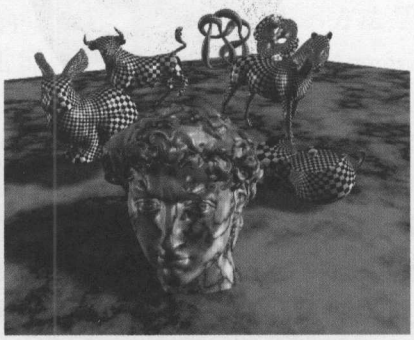


图 1-2　曲面纹理贴图

参数化不可避免地会带来畸变。通常，映射的畸变可以归为两类：角度畸变和面积畸变。角度没有畸变的映射，称为共形变换，或者保角变换；面积元没有畸变的映射，称为保面积变换。图 1-3 所示为一个保角参数化的实例。这一映射将一个三维人脸曲面映射到平面圆盘上。我们在人脸曲面任意画两条相交曲线，这两条曲面上的空间曲线被映射到平面上的两条曲线，空间曲线的交点被映成平面曲线的交点，在交点处，空间曲线的夹角等于平面曲线的交角。这两条空间曲线任意选取，其交角被映射完美保持。

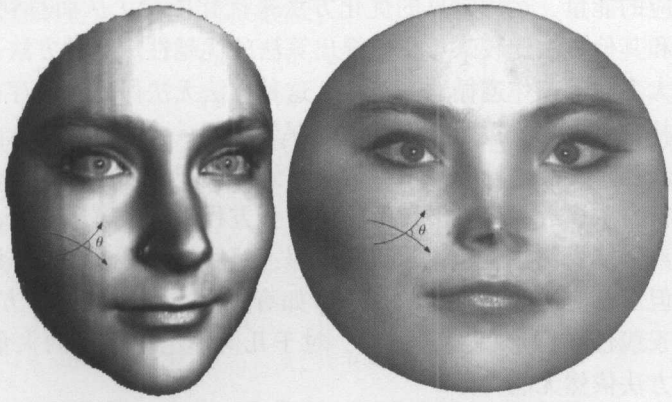


图 1-3　保角参数化实例

如图 1-4 所示为保面积参数化的一个实例。同样的人脸曲面被映射到平面圆盘，通过整体缩放变换，人脸曲面的总面积等于平面圆盘的总面积。我们在人脸曲面上任选一个子区

域 Ω，映射将 Ω 映射到平面一个区域，则曲面上的区域面积等于平面区域的面积。曲面参数化的一个长期目标就在于寻求严格而实用的算法，计算保角和保面积的映射。

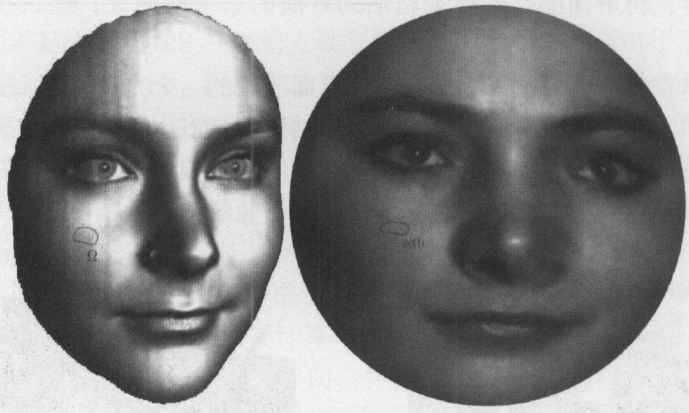


图 1-4　保面积参数化实例

在纹理贴图技术被发明之前，计算机图形学所需要的较深数学技巧包括矩阵求逆、旋转群的四元数表示（Quaternion）和渲染算法背后的积分方程、不动点理论；纹理贴图技术出现之后，曲面参数化需要更为深入的代数拓扑和微分几何的理论支撑。但是，绝大多数计算机科学系的基础课程都不包括代数拓扑和微分几何，这为年轻的学生理解跟踪曲面参数化领域的研究增加了难度。

2. 研究的范式

在过去的 20 年间，曲面参数化领域蓬勃发展，各种算法层出不穷，出现了百花齐放、百家争鸣的局面。海量的研究成果既有极其具有原创性的突破，也有拾人牙慧的跟风之作，更有人为运作的学术泡沫。这方面的工作需要长久的时间来蒸馏，20 年的时间依然太短。我们试图在各种流派中寻求统一的研究模式，并对于各种研究范式加以比较。

第一种研究范式比较工程化。首先将映射的质量要求表述成数学公式，然后通过积分这些公式来设计出相应的能量，运用常见的优化方法来优化能量，从而得到曲面参数化结果，最后选择几个算例和其他方法比较来彰显所提出算法的优越性。这种方法基于经验，简单直观，容易实现，绝大多数的工作遵循这一范式。这种方法无法保证解的存在性、唯一性和光滑性，缺乏全局拓扑观点，无法揭示几何现象的深刻层面。

第二种研究范式基于连续理论的离散逼近。首先深刻理解相应的经典几何理论；将光滑曲面、光滑映射进行离散逼近，将光滑情形的偏微分方程用离散解来逼近，或者将光滑情形的变分问题转化成离散能量优化问题，求得离散解；然后估计逼近误差，证明收敛阶。这种方法具有严格性，但需要应用数学方面的训练，如有限元方法、偏微分方程理论等。并且，这种方法比较适用于线性椭圆形偏微分方程，对于几何理论中出现的大量非线性偏微分方程，传统的有限元方法依然无能为力。

第三种研究范式直接建立离散的几何理论。传统的几何理论是建立在光滑流形基础之上的，需要曲面的微分结构，离散几何理论建立在多面体网格之上，然后证明离散曲面无限细分之后，离散理论导出光滑理论。这种研究范式对于非线性几何偏微分方程非常有效，同时给出了经典理论的一种全新理解方式。但是，因为缺乏成熟的数学工具（如第二种范式中

的伽辽金方法），这种研究具有很大的挑战性。

目前来看，在现实生活中，第一种研究范式的成果被工程类学术界广泛接受，并在工业界被广泛应用。但因为其缺乏严格性会被数学领域所拒绝；第二种研究范式和第三种研究范式的成果会被数学界接受，但因为其内容艰涩、逻辑曲折，很少会被工业界直接采纳。许多计算机科学家认为，那些被大型公司所采纳的算法会日渐胜出；许多数学家坚信，从长远来看，随着岁月流逝，大浪淘沙，具有普适性和严格理论保证的方法会被保留下来。

3. 工程化的研究方法

计算机只能表示离散数据，光滑的曲面一般用分片线性的多面体网格（Piecewise - linear Polyhedron）来逼近，又称三角网格（Triangle Mesh）、离散曲面（Discrete Surface），曲面之间的映射，用分片线性映射（Piecewise linear mapping）来逼近。这种表示方法是自然的，并且具有理论依据。在代数拓扑中有一条单纯逼近定理（Simplicial Mapping Approximation），大意是说任何连续映射，都可以在适当的三角剖分下，用单纯映射来逼近。给定任意的逼近精度要求，我们可以动态加细三角剖分，使得单纯映射满足精度要求。在理论和算法层面，几何逼近论着重研究下列问题：对于给定的光滑曲面和逼近精度要求，如何选取相应的三角剖分，构造离散的三角网格，使得在不同的度量意义下离散曲面和光滑曲面之间的“距离”小于给定阈值。这方面已经具有成熟的结果，如 Normal Cycle 理论。但对于给定光滑曲面间的光滑映射，并不存在统一系统的理论结果，绝大多数情况下依赖于所采用的方法，有各种各样的理论估计。

线性映射工程上对于曲面参数化问题研究的思路如下：曲面到平面区域的映射被表示成分片线性映射，通过对每一片线性映射进行分析，我们可以掌控整个映射。因此，我们首先来分析两个（欧式）平面三角形之间的线性映射，$\varphi:\mathbb{E}^2\to\mathbb{E}^2$。源欧式平面的坐标记为$(x,y)$，内积为

$$\langle (x_1,y_1),(x_2,y_2)\rangle - x_1x_2 + y_1y_2$$

目标欧式平面的坐标记为(u,v)，内积为

$$\langle (u_1,v_1),(u_2,v_2)\rangle = u_1u_2 + v_1v_2$$

通过平移，线性映射可以用矩阵来表示：

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = A\begin{pmatrix} x \\ y \end{pmatrix}$$

如果映射非退化，则矩阵满秩，$\mathrm{ranh}(\boldsymbol{A})=2$；如果映射保持定向，则矩阵行列式为正，$\det(\boldsymbol{A})>0$。下面的讨论中，我们假设 $\boldsymbol{A}$ 满秩，具有正值行列式。对于线性映射的细致分析，我们通常用矩阵的奇异值分解。

***QR* 分解**给定矩阵 $\boldsymbol{A}$，则 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 为对称正定矩阵，因此存在分解

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} \lambda_1^2 & \\ & \lambda_2^2 \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = O\mathrm{diag}(\lambda_1^2,\lambda_2^2)O^{\mathrm{T}}$$

这里 $\lambda_1,\lambda_2>0$ 为正实数。由此，我们可以定义 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 的平方根：

$$\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} \lambda_1 & \\ & \lambda_2 \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = O\mathrm{diag}(\lambda_1,\lambda_2)O^{\mathrm{T}}$$

显而易见：

$$\boldsymbol{A}=\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}\left(\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}\right)^{-1}\boldsymbol{A}=QR$$

这里 $Q=\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}$，同时 $R=\left(\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}\right)^{-1}\boldsymbol{A}$。我们直接计算

$$RR^{\mathrm{T}}=\left(\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}\right)^{-1}A\cdot A^{\mathrm{T}}\left(\sqrt{\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}}\right)^{-1}=I$$

因此矩阵 R 是旋转矩阵。因此矩阵 $A=QR$ 被分解为对称正定矩阵和旋转矩阵的乘积，这被称为矩阵的 QR 分解。

由矩阵的 QR 分解，我们直接得到矩阵的奇异值分解

$$A=QR=O_1(\lambda_1,\lambda_2)O_1^{\mathrm{T}}R=O_1(\lambda_1,\lambda_2)O_2$$

即矩阵可以被分解为旋转阵、对角阵、旋转阵的乘积。这里 λ_1,λ_2 称为奇异值，为正值实数。

线性映射分类根据矩阵奇异值，我们可以将线性映射分类。

（1）**等距变换**，如果 $\lambda_1=\lambda_2=1$，则矩阵为旋转阵，线性映射保持内积不变，换言之，$\langle \boldsymbol{r}_1,\boldsymbol{r}_2\rangle=\langle\varphi(\boldsymbol{r}_1),\varphi(\boldsymbol{r}_2)\rangle$。

（2）**相似变换**，如果 $\lambda_1=\lambda_2=\lambda$，则矩阵为旋转阵乘以一个标量，$A=\lambda O$。相似变换保持形状不变，因此可以被视为是共形变换的特例；相似变换保持角度不变，又被称为保角变换的特例。相似变换并不保持面积，其面积元变化率为 λ^2，

$$\mathrm{d}u\wedge\mathrm{d}v=\lambda^2\mathrm{d}x\wedge\mathrm{d}y$$

相似变换诱导的内积变换为

$$\langle\varphi(\boldsymbol{r}_1),\varphi(\boldsymbol{r}_2)\rangle=\lambda^2\langle\boldsymbol{r}_1,\boldsymbol{r}_2\rangle$$

（3）**保面积变换**，如果 $\lambda_1\lambda_2=1$，则矩阵为

$$A=O\mathrm{diag}(\lambda,\lambda^{-1})O^{\mathrm{T}}$$

面元保持不变，$\mathrm{d}u\wedge\mathrm{d}v=\mathrm{d}x\wedge\mathrm{d}y$。

我们可以看到，等距变换必为保面积变换；如果一个线性映射既是相似变换，又是保面积变换，则此变换必为等距变换。

（4）**拟共形变换**，一般情况下矩阵奇异值不等 $\lambda_1\neq\lambda_2$，源平面上取一个标准圆 $x^2+y^2=r^2$，我们考察其像

$$\begin{pmatrix}u\\v\end{pmatrix}=A\begin{pmatrix}x\\y\end{pmatrix}=O_1\begin{pmatrix}\lambda_1&\\&\lambda_2\end{pmatrix}O_2\begin{pmatrix}x\\y\end{pmatrix}$$

我们进行坐标变换

$$\begin{pmatrix}\xi\\\eta\end{pmatrix}=O_1^{-1}\begin{pmatrix}u\\v\end{pmatrix}$$

则标准圆的像为一椭圆

$$\frac{\xi^2}{\lambda_1^2}+\frac{\eta^2}{\lambda_2^2}=r^2$$

椭圆的偏心率由奇异值 λ_1,λ_2 所决定。如果椭圆为圆，则拟共形变换为共形变换。

（5）**变换的极分解**，矩阵的 QR 分解具有更为深刻的理解。假设 $A=QR$，这里矩阵 R 是旋转矩阵，可以视作保面积映射。构造映射

$$\tau:(x,y)\to(\xi,\eta)$$

$$\tau:\begin{pmatrix}\xi\\\eta\end{pmatrix}=R\begin{pmatrix}x\\y\end{pmatrix}$$

则 $d\xi \wedge d\eta = dx \wedge dy$。

Q 是一个正定矩阵，它代表了一个最优传输映射（Optimal Mass Transportation Map）

$$\rho:(\xi,\eta)\to(u,v)$$

$$\rho:\begin{pmatrix}u\\v\end{pmatrix}=Q\begin{pmatrix}\xi\\\eta\end{pmatrix}$$

更进一步，变换是某个凸函数的梯度映射 $\rho(\xi,\eta)=\nabla f(\xi,\eta)$，这里凸函数

$$f(\xi,\eta)=\frac{1}{2}(\xi,\eta)Q\begin{pmatrix}\xi\\\eta\end{pmatrix}$$

可以证明，映射 ρ 将面积元 $dx \wedge dy$ 映成面积元

$$du \wedge dv = \det(A)\,dx \wedge dy,$$

同时，在所有满足上述条件的映射中，ρ 极小化如下的传输代价：

$$E(\rho):=\int_{\Omega}|p-\rho(p)|^2 dxdy$$

这里的 Ω 是任意一个紧凸集。

非线性推广以上线性映射的分类结果可以（非平庸地）推广到曲面映射情形，基本想法如下。假设给定带度量的曲面间的光滑映射 $\varphi:(S_1,\boldsymbol{g}_1)\to(S_2,\boldsymbol{g}_2)$，局部上，非线性的映射 φ 可以用其一阶近似，切平面间的线性映射来逼近，所谓的导映射（Derivative Map）

$$d\varphi:T_pS_1\to T_{\varphi(p)}S_2$$

φ 会带来各种畸变，这些畸变的测量需要用到黎曼度量。线性映射 $d\varphi$ 的分类可以利用上述线性映射分类的想法，例如，如果对于任意一点 $p\in S_1$，$d\varphi$ 都是等距（或者保角、保面积、拟共形），则整体上 φ 是等距（或者保角、保面积、拟共形）。但是，最优传输映射（Optimal Mass Transportation Map）、映射的极分解、极值拟共形变换等，本质上是整体的，局部上的理解无法诠释其内在实质。相应的共形几何（保角映射）、最优传输（保面积映射）、泰希米勒理论（极值拟共形映射），我们会在后继的内容中阐述。

直接变分法在计算机图形学的曲面参数化领域，有许多工程化的研究方法，可以归为第一类研究范式。这种方法的基本思路如下：我们用分片线性映射来表示从三角网格到平面区域的映射，$\varphi:M\to\mathbb{D}$。固定一个三角形，我们将其铺到平面上，我们用 v_i,v_j,v_k 来表示顶点的平面坐标，用 $\varphi(v_i),\varphi(v_j),\varphi(v_k)$ 来表示顶点像的平面坐标，则线性映射矩阵可以直接写出：

$$A_{ijk}=(v_j-v_i,v_k-v_i)^{-1}(\varphi(v_j)-\varphi(v_i),\varphi(v_k)-\varphi(v_i))$$

这一映射和保角映射的差距可以定义为：

$$e_{ijk}:=\left(\frac{\lambda_1}{\lambda_2}+\frac{\lambda_2}{\lambda_1}-2\right)^2=\left(\frac{\lambda_1^2+\lambda_2^2}{\lambda_1\lambda_2}-2\right)=\left(\frac{\mathrm{tr}(A_{ijk})^2}{\det(A_{ijk})}-4\right)^2$$

如果 e_{ijk} 为 0，则限制在此三角形上的线性映射为保角变换。整个映射和等距变换的差距定义为：

$$E(\varphi):=\sum_{ijk}e_{ijk}s_{ijk}$$

这里 s_{ijk} 代表相应三角形的面积。然后，我们运用非线性优化方法来优化这个能量，并将所得结果作为全局保角变换的近似：

$$\varphi=\mathrm{argmin}_{\varphi}E(\varphi)$$

如果我们的目的是寻求保面积变换，我们可以如法炮制

$$e_{ijk} := (\lambda_1\lambda_2 - 1)^2 = (\det(A_{ijk}) - 1)^2$$

如果目的在于等距变换，那么我们可以定义局部能量

$$e_{ijk} := (\lambda_1 - 1)^2 + (\lambda_2 - 1)^2 = (\lambda_1 + 2)^2 - 2(\lambda_1 + \lambda_2) - 2\lambda_1\lambda_2 + 2$$

等价地

$$e_{ijk} := \mathrm{tr}(A_{ijk})^2 - 2\mathrm{tr}(A_{ijk}) - 2\det(A_{ijk}) + 2$$

在实际探索中，如何定义能量和如何优化能量成为研究的着力点。在理想情形下，我们希望能量为凸能量。

优缺点分析，这种工程化的研究方法比较简单直观，只需要线性代数的知识，很多时候也确实有效，因此在工程领域被广泛承认。同时，这种方法在理论方面存在缺陷：如果能量非凸，如何保证优化方法达到全局最优？即便是我们能够达到全局最优点，我们能够判定光滑情形的保角变换是否存在？如果真正存在光滑的保角变换，如何给出定量的误差分析？对于映射边界的控制如何，是否可以处理复杂拓扑曲面，对于所求映射的存在性、唯一性和光滑性，这种工程化的方法无法给出严格的答案。

从某种角度而言，线性代数的工具使得我们可以精确地描述问题、提出问题。**爱因斯坦曾经有一句名言，大意是对于一个真正有意义的问题，绝不可能在提出问题的层面解决，换言之，解决问题的层面要比提出问题的层面更为深刻。**因此，曲面参数化需要更为严密而深邃的理论支撑。

我们将在1.3节讲解第二种研究范式，包括调和映照理论及其几何有限元方法的近似。这种方法具有严格的理论基础，但无法涵盖更为复杂的非线性现象。

1.3 算法和理论（Ⅲ）

2017年1月11日，美国数学协会（American Mathematical Society）宣布2017年沃尔夫奖（Wolf Prize）授予Charles Fefferman和Richard Schoen。沃尔夫奖是数学领域的终身成就奖，Schoen是丘先生的弟子，获得沃尔夫奖名至实归。美国数学协会宣称Schoen在调和映照的正则性方面和Yamabe方程方面的研究影响深远。Schoen将调和映照从流形推广到抽象的度量空间，他有关Yamabe方程的工作，对于曲面参数化的研究产生了至关重要的影响。

计算机图形学中曲面参数化领域的许多算法是基于曲面调和映照理论（Harmonic Mapping），其内在原因包括诸多方面。

（1）**理论完备**调和映照理论历经数百年的发展，已经非常完善。调和映照的存在性、唯一性、光滑性都有艰深而精细的理论验证。

（2）**有效逼近**求解调和映照等价于求解椭圆形偏微分方程，数值上可以用有限元方法求解，解的收敛阶，误差估计都有经典结果。

（3）**数值稳定**根据椭圆型偏微分方程理论，其解连续依赖于边界条件。同时，线性椭圆形偏微分方程的有限元逼近是正定的线性系统，因此具有良好的数值稳定性。

（4）**普适性**经典调和映照理论适用于具有各种拓扑的曲面，现代调和映照理论适用于非流形的度量空间，如带度量的图（Metric Graph）。

（5）**微分同胚**在适当的拓扑和几何条件下，调和映照是光滑的微分同胚，这一点对于曲面参数化而言是至关重要的。

同时，相对于复杂的非线性方法，离散调和映照的**工程实现难度低**，容易上手开发。因此，在绝大多数的商用电影制作、游戏开发软件中，基于调和映照的参数化方法被广泛应用。

1. 几何直观

图1-5给出了从兔子曲面到单位球面的一个调和映射的实例。直观上，我们可以把兔子想象为由一张橡皮膜制成，映射将兔子曲面罩在光滑的大理石球体表面，橡皮膜和抛光的大理石表面间的摩擦力小到可以被忽略，因此橡皮膜可以在球面上自由滑动。这种自由滑动，在数学上可以用同伦变换来描述。无论兔子曲面如何滑动，映射的整体拓扑性质保持不变。直观上，兔子曲面被此映射罩在球面上，罩的代数层数就是整体拓扑不变量，其严格定义是所谓的映射“拓扑度”。同时，同伦变换会使得橡皮膜弹性形变的势能持续减小，到达平衡态时，弹性势能达到极小值，映射就是所谓的调和映照。因此，直观上，**调和映照极小化弹性形变的势能**。

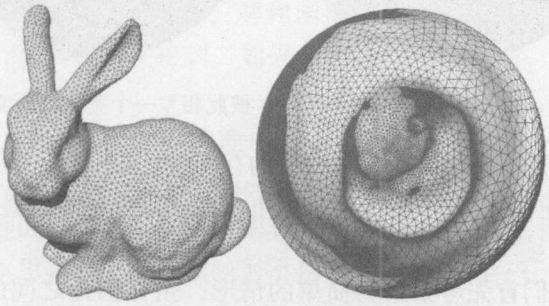


图1-5　兔子曲面的球面参数化

开始时，兔子曲面罩在球面上有可能有皱褶，局部上兔子曲面多层折叠。可以证明，经过弹性形变之后，所有的折叠都会被展开，所有的皱褶都会被熨平，最终整个兔子曲面均匀光滑地贴在球面上。数学上，这意味着**拓扑球面间的调和映照是微分同胚**。

更进一步讲，我们仔细观察图1-6所示小女孩雕塑到单位球面的调和映照，女孩的眉眼、口鼻、耳朵和发髻被映到球面上后，局部形状被完美保持。这意味着**拓扑球面间的调和映照是保角变换（共形变换）**。

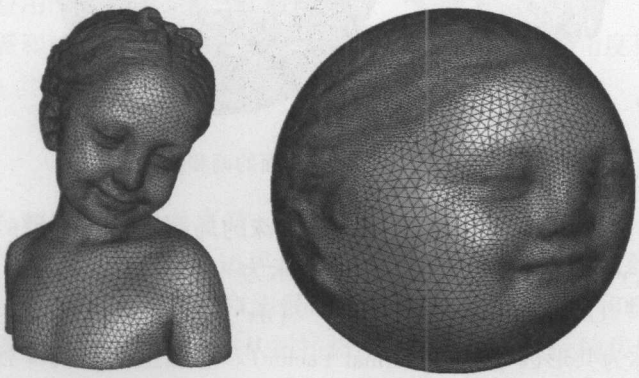


图1-6　小女孩雕塑的球面参数化

我们知道，通过球极投影，可以把单位球面保角地映射到拓展复平面上（复平面并上一个无穷远点），拓展复平面到自身的共形变换群为莫比乌斯变换群（Mobius Transformation Group）。这意味着**拓扑球面之间的调和映射不唯一，彼此相差一个莫比乌斯变换**。图 1-7 就显示了两个从小女孩雕塑到单位球面的调和映照，两个调和映照之间相差一个莫比乌斯变换和球极投影映射（及其逆映射）的复合映射。

图 1-7　球面调和映照不唯一，彼此相差一个莫比乌斯变换

由此，我们观察到了拓扑球面间调和映射的存在性、唯一性、正则性，以及和共形映射之间的关系。下面，我们探讨在不同拓扑情形下，调和映照的各种性质。

2. 拓扑圆盘

如图 1-8 所示，我们首先考察最为简单的情形，拓扑圆盘之间的调和映射 $\varphi:(S,\boldsymbol{g})\to\mathbb{D}^2$，这里曲面 S 是亏格为 0 的曲面，带有一条边界 ∂S。目标是平面上的单位圆盘 $\mathbb{D}^2$，其坐标为 (u,v)，映射 φ 可以被视作两个函数 $u,v:(S,\boldsymbol{g})\to\mathbb{R}$。

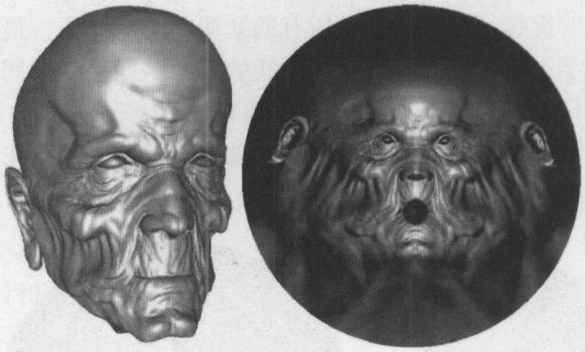


图 1-8　拓扑圆盘曲面的调和映射

为了简化讨论，我们可以采用曲面的一个特殊的局部坐标，所谓的等温坐标（Isothermal Parameters）(x,y)，使得黎曼度量张量被表示为

$$\boldsymbol{g}(x,y)=\mathrm{e}^{2\lambda(x,y)}(\mathrm{d}x^2+\mathrm{d}y^2)$$

这里 $\lambda:S\to\mathbb{R}$ 称为共形因子（Conformal Factor）。等温坐标的局部存在性曾经被陈省身先生用拟共形几何证明过。

在等温坐标下，函数的调和能量可以被定义为

$$E(u):=\int_S \langle \nabla u, \nabla u\rangle \mathrm{d}x\mathrm{d}y$$

调和函数极小化调和能量，其对应的欧拉－拉格朗日方程是

$$\Delta u=\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)u=0$$

给定边值条件 $u\mid \partial S=g$，调和函数满足拉普拉斯方程

$$\begin{cases}\Delta u=0\\ u\mid \partial S=g\end{cases}$$

根据经典的椭圆形偏微分方程理论，如 Riesz 表示定理，拉普拉斯方程的解存在。同时，调和函数满足**均值性质**（Mean Value Property）：给定一点 $p\in S$，一条环绕 p 点的曲线 $\gamma\subset S$，

$$u(p)=\frac{1}{|\gamma|}\oint_\gamma u(q)\mathrm{d}q$$

由均值性质，我们得到**极大值原理**，调和函数的极大值（极小值）一定取在边界上。调和函数光滑性，可以由 Weyl 引理得到。如果曲面边界足够光滑，边值条件足够光滑，则调和函数是光滑的。

调和映射的微分同胚性质由 **Rado 定理**来保证：如果 $\varphi:(S,\boldsymbol{g})\to\mathbb{D}^2$ 是调和映射，其在边界上的限制 $\varphi\mid\partial S:\partial S\to\partial\mathbb{D}^2$ 是拓扑同胚（Homeomorphism），则在内部映射为微分同胚（Diffeomorphism）。这一定理的证明充分利用了调和函数的极大值原理，同时利用了莫尔斯理论（Morse Theory）。其基本思路如下：假设映射不是微分同胚，则存在一个内点 $p\in S$，映射的 Jacobi 矩阵退化，存在常数

$$a,b\in\mathbb{R}$$

$$a\nabla u(p)+b\nabla v(p)=0$$

因此 $f=au+bv$ 为调和函数，p 点为 f 的奇异点，由极大值原理，p 点不可能是极大值或极小值，必为鞍点（Saddle Point）。那么，p 点附近的等值线（Level Set）有两个分支，两个分支和曲面的边界有 4 个交点。另一方面，f 在单位圆盘 $\mathbb{D}^2$ 上的等值线是直线段，和 $\mathbb{D}^2$ 边界有两个交点。这意味着曲面边界上的 4 个点，被映射到圆盘边界上的两个点，因此 $\varphi\mid\partial S:\partial S\to\partial\mathbb{D}^2$ 不是拓扑同胚。由此导出矛盾，假设错误。调和映射的微分同胚性质，是这种参数化方法被广泛应用的根本原因之一。

共形映射必为调和映射，但调和映射未必是共形映射。从分析角度而言，如果两个调和函数 $u,v:(S,\boldsymbol{g})\to\mathbb{R}$ 共轭，即它们满足柯西－黎曼方程：

$$\begin{cases}u_x=v_y\\ u_y=-v_x\end{cases}$$

那么调和映射是共形映射。几何上来看，我们选择 3 个边界点 $p_1,p_2,p_3\in\partial S$，固定它们的像，$\varphi(p_k)\in\partial\mathbb{D}_2$，沿着 $\partial\mathbb{D}_2$ 滑动其他边界点的像点，然后计算调和映射。如此变化边界条件，使得调和能量进一步减小，当调和能量达到最小时，所得的调和映射为共形映射。由此，我们得到了**黎曼映照定理**，即存在从拓扑圆盘曲面到单位圆盘的共形映射。

利用黎曼映照定理，我们可以极大地简化理论层面的讨论。首先，调和能量在源曲面的共形变换下不变，因此调和函数（调和映射）在源曲面的共形变换下不变。我们用黎曼映

照，将曲面保角地映到平面圆盘，那么曲面上的调和函数变成了平面圆盘上的调和函数，这时调和函数的解可以用边值条件和泊松核（Poisson Kernel）的卷积来直接写出，泊松核定义为

$$P_r(\theta) = \mathrm{Re}\left(\frac{1+e^{i\theta}}{1-e^{i\theta}}\right)$$

调和函数的公式为

$$u(re^{i\theta}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} P_r(\theta)g(e^{i\tau})\,\mathrm{d}\tau$$

这一公式显式地给出了调和映射解的存在性、唯一性和正则性证明。

对于一般的散度型的椭圆偏微分算子

$$a(x,y)\frac{\partial^2}{\partial x^2} + 2b(x,y)\frac{\partial^2}{\partial x\partial y} + c(x,y)\frac{\partial^2}{\partial y^2} + d(x,y)\frac{\partial}{\partial x} + e(x,y)\frac{\partial}{\partial y} + f(x,y)$$

这里矩阵

$$\begin{pmatrix} a(x,y) & b(x,y) \\ b(x,y) & c(x,y) \end{pmatrix}$$

处处正定，我们可以从几何上加以讨论。从几何上讲，我们可以找到一个定义在平面参数区域 Ω 上的黎曼度量 $\boldsymbol{g}$，使得椭圆形微分算子是度量 $\boldsymbol{g}$ 下的 Laplace 算子。进一步，我们可以找到度量 $\boldsymbol{g}$ 的等温坐标

$$\boldsymbol{g} = e^{2\mu(\xi,\eta)}(\mathrm{d}\xi^2 + \mathrm{d}\eta^2)$$

则微分算子成为标准的 Laplace 算子

$$\frac{1}{e^{2\mu(\xi,\eta)}}\left(\frac{\partial^2}{\partial\xi^2} + \frac{\partial^2}{\partial\eta^2}\right)$$

这意味着**散度型椭圆偏微分方程可以转化成经典的 Laplace 方程，只不过是变换了度量和边值条件。**

3. 拓扑复杂曲面

对于拓扑复杂的曲面，调和映照的理论也非常完备。给定度量曲面之间的映射 $u:(M,\boldsymbol{g})\to(N,\boldsymbol{h})$，我们采用复等温坐标，$\boldsymbol{g}(z) = \lambda(z)\mathrm{d}z\mathrm{d}\bar{z}$，并且 $\boldsymbol{h}(w) = \rho(w)\mathrm{d}w\mathrm{d}\bar{w}$。我们定义微分算子

$$\partial_z = \frac{1}{2}(\partial_x - \sqrt{-1}\partial_y),\quad \partial_{\bar{z}} = \frac{1}{2}(\partial_x + \sqrt{-1}\partial_y)$$

由此定义

$$|\partial u|^2 = \frac{\rho(u(z))}{\lambda(z)}|u_z|^2,\quad |\bar{\partial}u|^2 = \frac{p(u(z))}{\lambda(z)}|u_{\bar{z}}|^2$$

那么映射的调和能量密度为

$$e(u;\lambda,\rho) = |\partial u|^2 + |\bar{\partial}u|^2$$

映射的调和能量为

$$E(u;\lambda,\rho) := \int_M e(u;\lambda,\rho)\lambda\mathrm{d}x\mathrm{d}y = \int_M \rho(u(z))(|u_z|^2 - |u_{\bar{z}}|^2)\mathrm{d}x\mathrm{d}y$$

由此，我们可以看到**调和能量只和源曲面的共形结构有关，和具体的共形黎曼度量无关。**由此，我们得到调和能量的欧拉－拉格朗日方程为

$$(\log\rho)_u u_z u_z + u_{zz} = 0$$

内蕴的非线性热流方程为

$$\frac{\partial u(z,t)}{\partial t} = -\left(\frac{\rho_u(u)}{\rho(u)} u_{\bar{z}} u_z + u_{z\bar{z}}\right)$$

调和映射和共形映射的关系我们定义曲面间映射所诱导的二次微分，霍普夫微分：

$$\Phi(u) = \rho(u) u_z \overline{u}_{\bar{z}} \mathrm{d}z^2$$

我们可以证明，**如果霍普夫微分为全纯二次微分，则映射必为调和映射；如果霍普夫微分为0，则映射必为共形映射**。由此我们看到，共形映射必为调和映射，反之则不然。但是，拓扑球面上所有的全纯二次微分必然为0。因此，拓扑球面间的所有的调和映射，其霍普夫微分为全纯二次微分，必然为0。我们得到拓扑球面间的所有调和映射必为共形映射。

调和映射的微分同胚性丘成桐先生曾经证明过如下定理。**如果源曲面和目标曲面同为亏格为 g 的封闭曲面，调和映射的度为1，目标曲面上的曲率处处为负，那么调和映射必为微分同胚**。

我们用反证法给出简略证明。假设在某一内点，雅克比行列式为负，因此区域 $D = \{p \in S \mid J(p) < 0\}$ 非空。研究函数

$$\Delta_M \log \frac{|\partial u|}{|\bar{\partial} u|} = -4K_N J(u)$$

则它在 D 的边界上为0，在 D 的内部处处为负，因此函数 $\log|u_z|/|u_{\bar{z}}|$ 为上调和函数(Super Harmonic)，在 D 的内部处处为正，那么雅克比行列式

$$J(u) = |u_z|^2 - |u_{\bar{z}}|^2$$

在 D 的内部处处为正，这和 D 的定义相矛盾。

调和映射的唯一性。调和映射的唯一性和目标曲面的曲率具有非常紧密的联系。**封闭曲面间的调和映射，如果目标曲面上的高斯曲率处处为负，且 M 在 N 上的像不是一条闭测地线，则同伦的调和映射必然重合**。一种想法是基于调和能量的凸性，设定二阶光滑同伦连接着两个调和映射

$$F: M \times [0,1] \to N, u_t(x) = F(x,t), \quad \forall x \in M$$

我们计算调和能量的二阶导数

$$\frac{\mathrm{d}^2 E(u_t)}{\mathrm{d}t^2} = 2\int_M \left(\sum_i |\nabla_{e_i} V|^2 - K_N(V, F_* e_i) + \langle \nabla_{e_i} \nabla_{\frac{\partial}{\partial t}} V, F_* e_i \rangle^2\right) \mathrm{d}A$$

这里切矢量场定义为 $V = F_* \partial/\partial t$，联络算子 ∇，源曲面上的标准正交基为 $\{e_1, e_2\}$，目标曲面上的高斯曲率为 K_N。我们选取测地同伦，也即 $\forall x \in M, t \to u_t(x)$ 为测地线，则

$$\nabla_{\frac{\partial}{\partial t}} V = 0$$

那么最后一项恒为0。因此调和能量的二阶导数恒正，调和能量为严格凸。同时在时间为0和1点处，映射为调和映射，调和能量关于时间的一阶导数为0，由此我们有

$$\frac{\mathrm{d}^2 E(u_t)}{\mathrm{d}t^2} \equiv 0, |\nabla_{e_i} V| \equiv 0, \quad K_N(V, F_*, e_i) \equiv 0$$

并进一步我们得到 V 必然处处为0，因此起始和终点处的调和映射重合。

调和映射的存在性。根据调和映射理论，**如果两个曲面间有一个微分同胚，那么存在一个和此微分同胚同伦的调和映射**。如果曲面间有一个非同胚的映射，那么是否一定存在一个

与之同伦的调和映射呢？这个问题的答案是否定的。调和映射的存在性被整体拓扑条件所限定。如果辅助函数$|\partial u|$，$|\bar{\partial} u|$不恒为0，那么辅助函数零点的总阶数和曲面的拓扑及映射的映射度之间有着整体的关系

$$\sum_{|\partial u(p)|=0} n_p = -\deg(u)(4g_N - 4) + (4g_M - 4)$$

$$\sum_{|\bar{\partial} u(p)|=0} n_p = +\deg(u)(4g_N - 4) + (4g_M - 4)$$

这一关系制约着调和映照的存在。

调和映射的推广。调和映照可以被推广到非流形情形，如更为一般的度量空间。其中有一种特例非常重要，从曲面到带度量的图（Metric Graph）的调和映射。图的万有覆盖空间是树（Tree），树上任意两点之间的最短路径唯一。我们知道，高斯曲率为负的曲面上，任意两点之间，每一个同伦类中存在唯一的测地线。由此，我们可以想象树的曲率为负，图的曲率为负。Gromov 和 Schoen 定义了图的双曲性质，证明了从曲面到图的调和映照的存在性和唯一性，其诱导的霍普夫微分也是全纯二次微分。

4. 有限元法

传统的伽辽金方法就是在曲面的泛函空间中构造有限维子空间，将曲面任意函数向子空间投影，用这一投影来近似原函数。随着子空间维数的增加，近似函数趋近原函数。有限元方法将曲面三角剖分，每一个三角形用欧式三角形来近似，形成三角网格。曲面上的函数用网格上的分片多项式函数来近似。有限元法将椭圆形偏微分方程（如 Lapalce 方程）转换成等价的变分形式（调和能量优化），将线性偏微分方程转化为线性方程组来求解。在曲面参数化算法中，通常人们应用分片线性元，用分片线性函数来进行近似。给定一个三角形$[v_i, v_j, v_k]$，给定线性函数$f:[v_i, v_j, v_k] \to \mathbb{R}$，通过直接计算，我们得到函数在三角形上的调和能量为

$$E(f) = \cot\theta_i (f_j - f_k)^2 + \cot\theta_j (f_k - f_i)^2 + \cot\theta_k (f_i - f_j)^2$$

整个三角网上的调和能量为

$$E(f) = \sum_{[v_i, v_j]} w_{ij} (f_i - f_j)^2$$

这里w_{ij}被称为余切权重，是边$[v_i, v_j]$所对的两个角的余切之和。通过对调和能量求导，我们得到离散拉普拉斯算子

$$\Delta f(v_i) = \sum_{[v_i, v_j]} w_{ij} (f(v_j) - f(v_i))$$

由$\Delta f(v_i) = 0$，我们得到离散调和函数的均值性质

$$f(v_i) = \frac{\sum_j w_{ij} f(v_j)}{\sum_k w_{ik}}$$

即每个顶点的值等于其相邻顶点值的加权平均。由此，我们可以得到离散调和映射的迭代算法：在每一次迭代中，我们将每个顶点的像移到与其相邻顶点像的加权重心。

Rado 定理的离散版本也成立，从拓扑圆盘到平面凸区域的离散调和映射，如果边界映射是拓扑同胚，则内部也是拓扑同胚。

有限元法需要强有力的网格生成算法，对所用网格质量有一定的要求。对于平面区域，最为通用的方法当推 Delaunay Refinement 算法，如果$\partial\Omega$不存在过于尖锐的内角，则三角剖

分的最小内角可以被保证。如果三角剖分质量达到要求，则离散解收敛到连续解，考虑函数值和一次导数，解的误差界为 $O(\varepsilon)$，这里 ε 为三角形边长。

5. 高维推广

调和映射方法在工业中被广泛应用，这可以归功为完备的连续理论和离散理论，以及简单稳定的算法。同样的想法可以应用于三维体的参数化（Volumetric Parameterization）。但是，当将曲面调和映射向三维推广时，我们遇到了本质困难：**Rado 定理在三维不成立**。

如图 1-9 所示，我们用曲面的调和映射将佛像曲面映到单位球面上，这一映射是保角的，因此是同胚映射。我们以这一映射为边界条件，计算佛像内部到球体内部的调和映射，如图 1-10 所示，理论上无法保证体调和映射一定是同胚映射。这是体参数化和曲面参数化的本质区别之一。如何保证体映射是同胚，这是当前参数化领域的主要热点问题。

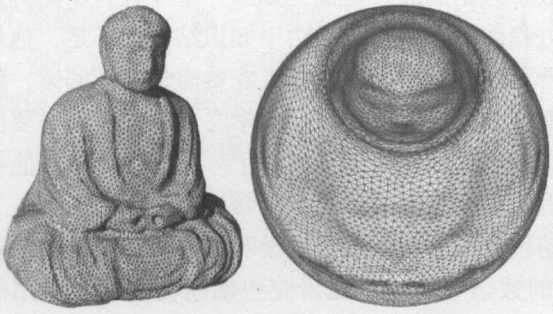


图 1-9　佛像曲面的球面参数化

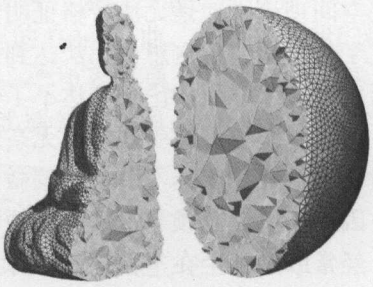


图 1-10　体内调和映射

6. 小结

曲面间的调和映照问题等价于在曲面上求解几何偏微分方程问题，解的存在性、唯一性、正则性强烈依赖于拓扑条件（亏格、同伦类、映射度等）和几何条件（黎曼度量的高斯曲率）。很多理论结果非常令人惊异（惊艳），比较反直觉，如拓扑球面间的调和映照一定是共形映射，到负曲率空间的调和映照唯一并且是微分同胚等。理论比较艰深，需要黎曼几何、几何偏微分方程等领域的知识。

从计算方法角度而言，离散调和映射的技术非常直观简单：固定初始映射的同伦类之后，循环迭代，局部减小调和能量。绝大多数的研究生都可以自己提出并实现这种算法。但是，工程的观点聚焦于局部优化，缺乏整体大局观，整体解的性状需要理论支撑。

目前，很少有学校的课程涵盖调和映照理论，绝大多数时在基础数学的讨论班中有所讨论。但是，离散调和映射的算法，几乎所有计算机图形学的研究生课程都会涉及。从这个角度而言，计算机技术的普及将纯粹数学请出了象牙塔，极大地促进了现代数学的传播和发展。从另一个角度讲，目前调和映照的算法只涉及了调和映照理论的极小部分，依然有广阔的空间等待抽象理论到具体算法的实现。

7. 观察

在过去的 10 多年间，笔者老顾在很多大学开设“计算共形几何”课程，遇到了许多学者、学生。大多数计算机背景的青年学生比较务实，有些学生急于获得学位投身到信息工业之中，因此对于算法非常热衷，对于理论比较冷淡；更有很多学生，他们具有强烈的好奇心和旺盛的求知欲，他们并不满足“知其然”，更加追求“知其所以然”。在北美、香港、北

京、长春，笔者老顾经常遇到这些学生，他们承受着巨大的生活和学业压力，但依然对艰深的理论孜孜以求。他们对于精粹而无功用学问的追求和当下主流价值观念背道而驰。很多时候，他们的理念不被同学甚至导师所理解，他们和周围的环境格格不入。他们的坚守和执著非常令笔者老顾感动。

从以上的讨论我们看到，调和映照的理论需要很多非常深入的数学知识、椭圆形偏微分方程、微分几何、代数拓扑、黎曼面理论、黎曼几何等。在基础数学领域，学生们需要花费七八年的时间才能完成这些课程，做到融会贯通、掌握精髓更加靡费时日；对于计算机科学背景的学生而言，自学这些课程几乎是不可能的事情。笔者老顾和清华大学数学学院的肖杰院长曾经交流过，肖教授倾向于认为代数可以自学成才，但分析只能采用传统的师徒学制才能深刻掌握。笔者老顾曾经观察过许多非常有才华的计算机背景的学生，他们不满足于经典计算机科学的内容，自行探索更为深刻的数学理论层面。但是，如果一个年轻人不懂得索博列夫空间理论，他是无法严格证明偏微分方程解的存在性的。由于知识结构的不足，他们经常耽于哲学层面的空想，无法达到公认的严格水平。他们中很多人花费海量时间阅读，但只学习工程技术方面的论文到达不了他们所期待的理论深度，阅读数学文章，又缺乏必要的功力，很多时候他们饱受挫折，无奈放弃。纵然今天的网络几乎连接了一切，但网络只能提供相对肤浅的碎片化学习，对于偏微分方程理论这个层次的学习似乎帮助不大。

因此，笔者老顾非常赞同跨领域培养的教育模式，如结合数学和计算机科学的加强班制度；经常鼓励学生在本科期间多学习纯粹的数学和物理，在研究生期间再转相对实用的方向。笔者老顾经常访问法国里昂的大学，和计算机系的陈立铭教授探讨过教育问题。陈教授介绍说，在法国的教育系统中，高中生经过苛刻的选拔后，精英才俊会进入特殊的大学，在那里他们四年时间只接受数学和物理方面的训练。这是法国数学一直傲视群雄的核心原因之一，也是法国航空工业发达强大的原因。（迄今为止，波音飞机的设计实际是用法国的 CAD 软件。）

今天的中国，物质文明日益强大，必将涌现精神贵族阶层。笔者老顾相信会有越来越多的青年才俊，不再为谋生所迫，遵从内心召唤，大胆追求深刻理论的美学价值，为人类的精神文明做出杰出贡献。

在 1.4 节，我们会给出共形几何理论、泰希米勒空间理论和最优传输理论的概览，这些理论构成了曲面参数化的理论基础，其本身具有强烈的美学价值。

1.4 算法和理论（Ⅳ）

计算机图形学中的曲面参数化方法可以看成微分几何中曲面映射理论的直接应用。就目前情况而言，理论方面的发展远远超过工程方面的发展，有大量的理论成果还没有被转化成实用的算法；从另一方面而言，理论方面的关注点在于研究各种映射的存在性、唯一性，解空间的结构，因此停留在抽象层面，工程方面的重点在于构造和逼近各种映射，精密而具体。下面，我们简略讨论一下参数化相关的理论。

1. 拓扑层面

假设 S 是一个高亏格曲面，我们考察所有曲面到自身的拓扑同胚 $\varphi: S \to S$，显然在映射的复合作用下，所有的曲面自同胚构成一个群。如果我们将所有的自同胚进行同伦分类，则

所有的自同胚同伦类构成一个群，记为曲面的映射类群（Mapping Class Group）MCG(S)。

我们首先考察圆柱面的情形，$A=\mathbb{S}^1\times[0,1]$，如图1-11所示，Dehn扭曲是一个保持边界不动的映射

$$T:A\to A,\quad T(\theta,t)=(\theta+2\pi t,t)$$

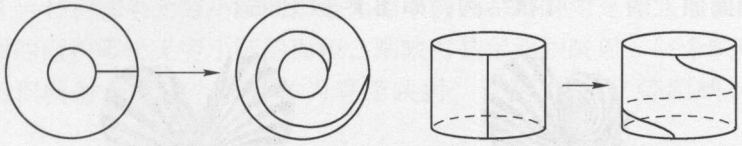


图1-11　Dehn扭曲

假设α是曲面S上一条简单非平庸的闭曲线，简单意味着曲线不自相交，非平庸意味着曲线不能缩成一点，则α在曲面上的一个邻域A_α是拓扑柱面。我们构造曲面的自同胚T_α:$S\to S$，使得T_α在A_α之外是恒同映射，T_α限制在A_α上是如上的Dehn扭曲。直观上，我们将S沿着曲线α切开，这样新产生两个边界联通分支，然后将一个边界联通分支的邻域拧转2π角度，再将两个边界联通分支重新黏合，如此得到的映射就是T_α，如图1-12所示。

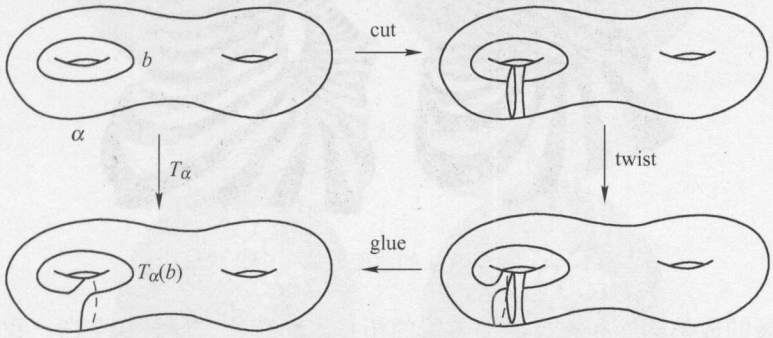


图1-12　曲面上关于简单闭曲线α的Dehn扭曲

实际上，沿着不同的简单非平庸的圈的Dehn扭曲生成了曲面映射类群。Humphries在1979年证明了亏格为$g\geqslant 3$的闭曲面上$2g+1$条曲线对应的Dehn扭曲就足以生成MCG(S)。如图1-13所示，简单非平庸闭曲线$\{c_0,c_1,\cdots,c_{2g}\}$对应的Dehn扭曲是群MCG($S$)的生成元。

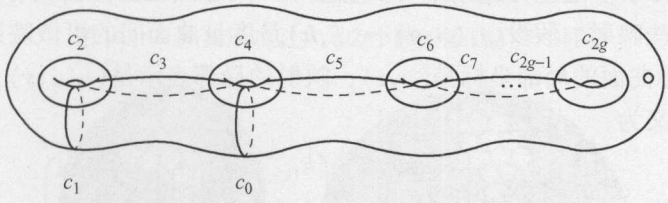


图1-13　Humphries Dehn扭曲生成元

瑟斯顿（William Thurston）系统地发展了曲面映射分类理论。所有的曲面自同胚同伦类被分成3种标准类型。

（1）周期的：存在一个正整数n，使得φ^n和恒同映射同伦。

（2）Anosov：存在曲面上的两个叶状结构（Foliation），映射φ保持这两个叶状结构不

变。所谓的叶状结构就是将曲面（去除几个奇异点）分解成一维曲线的集合。

（3）可分解：曲面上存在有限个简单闭曲线$\{c_0,\cdots,c_k\}$，它们在映射下的像$\{\varphi(c_0),\cdots,\varphi(c_k)\}$是原来曲线的一个重排列。曲面被这些曲线切割，映射在每个联通分支上的限制都是 Anosov 的。

亏格为 3 的曲面上的一个叶状结构，如图 1-14 所示。

图 1-14　亏格为 3 的曲面上的一个叶状结构

和理论发展的成熟程度相比，曲面映射的拓扑工程算法方面一片空白。我们几乎难以找到判别曲面映射同伦类，或者判别曲面映射的 Thurston 分类的算法。工程的发展依然严重滞后于理论的发展。

2. 几何层面

我们在 1.2 节中，给出了线性映射大致分类：等距变换、相似变换、保面积变换、拟共形变换、变换的极分解。这些分类结果可以直接推广到非线性的曲面映射情形。我们用黎曼度量来精确定义这些映射。假设$\varphi:(S,\boldsymbol{g})\to(T,\boldsymbol{h})$是度量曲面间的可微映射，源曲面的局部坐标是$(x,y)$，目标曲面的局部坐标是$(u,v)$，映射的局部表示是$(u(x,y),v(x,y))$。映射$\varphi$诱导的拉回度量定义为：

$$\varphi^{*}\boldsymbol{h}=(\mathrm{d}x\quad \mathrm{d}y)\begin{pmatrix}u_x & v_x\\ u_y & v_y\end{pmatrix}\begin{pmatrix}h_{11} & h_{12}\\ h_{21} & h_{22}\end{pmatrix}\begin{pmatrix}u_x & u_y\\ v_x & v_y\end{pmatrix}\begin{pmatrix}\mathrm{d}x\\ \mathrm{d}y\end{pmatrix}$$

直观上，拉回度量可以如下理解：假设$\gamma:[0,1]\to S$是源曲面上的一条曲线，映射φ将其映射到目标曲面$\varphi(\gamma)\subset T$，我们用目标曲面上的黎曼度量$\boldsymbol{h}$来测量$\varphi(\gamma)$的长度，用这个长度来定义γ的长度，如此得到的度量就是映射φ诱导的拉回度量。我们用拉回度量来定义各种映射。

（1）等距映射：拉回度量等于源曲面的度量$\boldsymbol{g}=\varphi^{*}\boldsymbol{h}$；

（2）共形映射：存在函数 $\lambda: S \to \mathbb{R}$，使得 $\varphi^* \boldsymbol{h} = e^{2\lambda} \boldsymbol{g}$，即拉回度量和初始度量相差一个正的标量函数；

（3）保面积映射：拉回度量的行列式等于初始度量的行列式，$\det(\varphi^* \boldsymbol{h}) = \det(\boldsymbol{g})$。

另外一种等价的直观解释是考察无穷小圆的像：如果无穷小圆的像是无穷小圆，则映射是共形映射，见图 1-15；如果无穷小圆的像是无穷小椭圆，则映射是拟共形映射，见图 1-16；如果无穷小椭圆的面积等于无穷小圆的面积，则映射是保面积映射。一个映射如果既是共形映射，又是保面积映射，则这个映射必为等距映射，见图 1-17。等距映射保持高斯曲率不变。

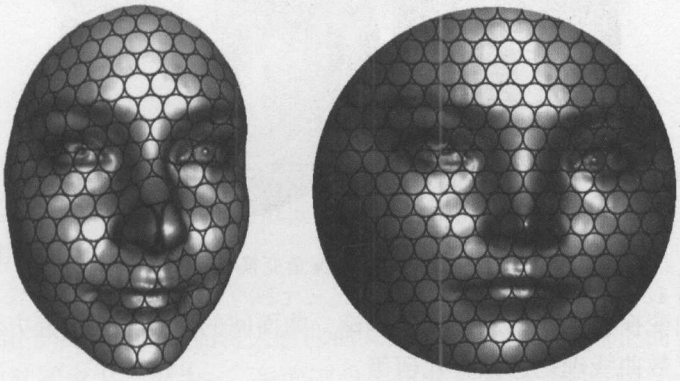


图 1-15　共形映射：无穷小圆的像是无穷小圆

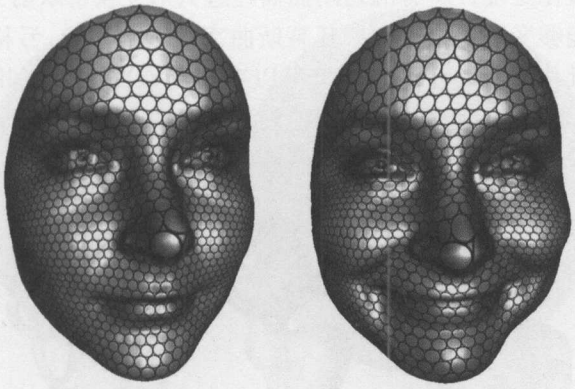


图 1-16　拟共形映射：无穷小圆的像是无穷小椭圆

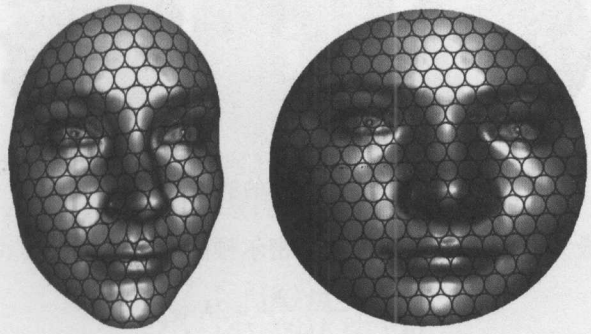


图 1-17　保面积映射：无穷小椭圆的面积等于相应的无穷小圆的面积

拓扑等价的紧曲面之间不一定存在共形映射，但一定存在拟共形映射，所有拟共形映射中最为接近共形映射的是所谓的 Teichmuller 映射。我们下面简单介绍各种映射的几何理论。

3. 共形映射

共形映射（Conformal Mapping）又称保角变换（Angle - preserving Mapping），如图 1-18 所示：我们在曲面上选择两条相交曲线，它们被映射成平面上的两条相交曲线，交角保持不变。

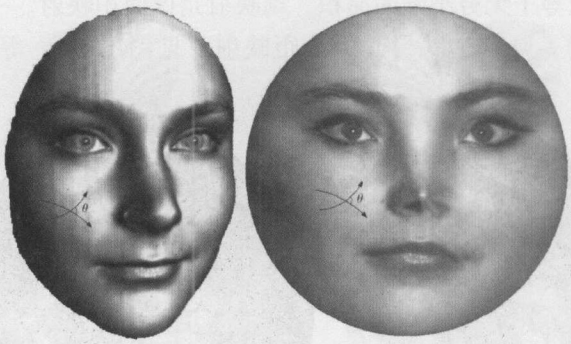


图 1-18　保角变换

平面上的保角变换主要应用复变函数理论，曲面间的保角变换研究方法非常丰富，主要是黎曼面理论、代数曲线理论、微分几何等。

共形映射最为深刻，也是最为直接的定理即所谓的单值化定理（Uniformization）。图 1-19 显示了封闭曲面的单值化定理，所有的封闭曲面经过共形变换都映射到三种标准曲面中的一种：亏格为零的曲面能够变成单位球面，其高斯曲率处处为 +1；亏格为一的曲面能够变成平环，其高斯曲率处处为 0；高亏格的曲面可以变换为双曲曲面，其高斯曲率处处为 -1。图 1-20 显示了带有边界的曲面的单值化定理，这时所有的边界都变成了圆周。单值化定理从理论上保证了现实生活中所有曲面的全局参数化。

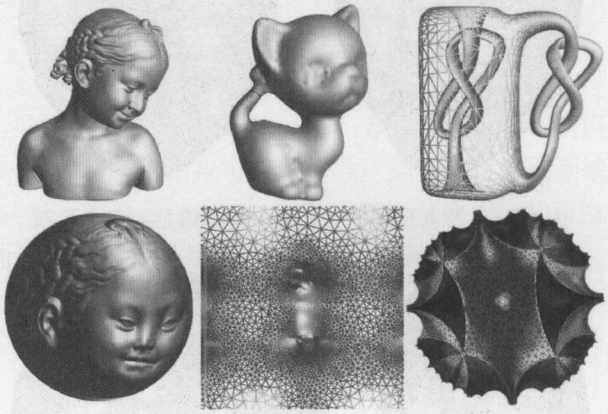


图 1-19　封闭曲面的单值化定理

曲面单值化定理最为有效的计算方法是哈密尔顿的 Ricci 流

$$\frac{\mathrm{d}g_{ij}}{\mathrm{d}t}=\left(\frac{4\pi\chi(S)}{A(0)}-2K\right)g_{ij}$$

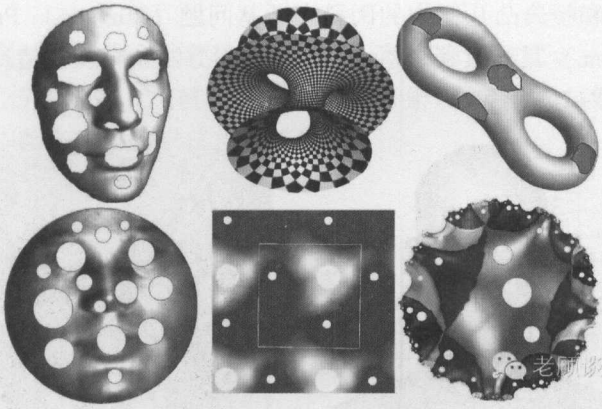


图 1-20　带边界曲面的单值化定理

Ricci 流理论将曲面的黎曼度量形变，使得曲率像热流那样扩散，直至处处成为常数。曲率流的理论已经从光滑曲面推广到离散曲面情形，目前相对成熟。

4. 保面积映射

图 1-21 显示了保面积映射的一个例子，三维曲面上的任意可测集 Ω 被映射到平面区域 $\varphi(\Omega)$，两者面积相等。首先，我们将人脸曲面共形地映到平面圆盘上面，这样我们将人脸曲面上的面积元映到圆盘上，记为 μ。平面圆盘上本来具有欧氏的面积元，记为 ν。我们能够找出一个特殊的映射 $T:(\mathbb{D}^2,\mu)\to(\mathbb{D}^2,\nu)$，满足两个条件：

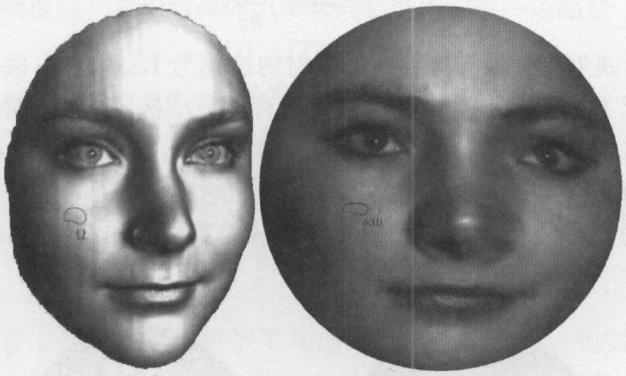


图 1-21　保面积映射示例

（1）映射将概率测度 μ 映射成概率测度 ν。

（2）映射极小化下面的传输代价：

$$E(T):=\int_{\mathbb{D}^2}|x-T(x)|^2\mathrm{d}\mu(x)$$

这种映射被称为最优传输映射。根据最优传输映射的 Kontarovich - Brenier 理论，存在一个凸函数 $u:\mathbb{D}^2\to\mathbb{R}$，其梯度映射就是最优传输映射，$T(x)=\nabla u(x)$。由此，这个特殊的函数满足蒙日 - 安培方程：

$$\det\left(\frac{\partial^2 u}{\partial x_i\partial x_j}\right)=\frac{\mu}{\nu\circ\nabla u}$$

蒙日－安培方程和经典凸几何中的闵科夫斯基问题（Minkowski Problem）、亚历山大问题（Alexandrov Problem）具有紧密联系，可以用几何方法来直接构造。图 1-22 显示了这种方法得到的保面积映射。

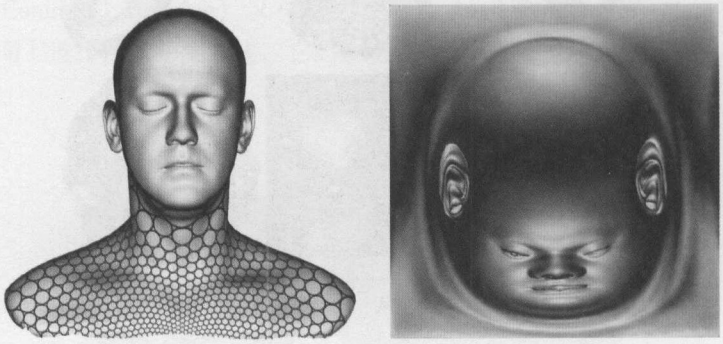


图 1-22　保面积映射效果

5. 拟共形映射

一般曲面间的微分同胚都是拟共形映射（Quasi－conformal Mapping），每个无穷小圆被映射到无穷小椭圆。无穷小椭圆的偏心率（长短轴之比）和长轴方向信息构成所谓的 Beltrami 系数 μ，Beltrami 系数完全控制曲面的映射。给定 Beltrami 系数，我们可以通过求解 Beltrami 方程来重建映射

$$\frac{\partial \varphi}{\partial \bar{z}}=\mu(z)\frac{\partial \varphi}{\partial z}$$

在所有的拟共形映射中，最为接近共形映射的被称为 Teichmuller 映射。Teichmuller 映射使得无穷小椭圆的上界达到最小，其特征就是所有无穷小椭圆的偏心率都相等，如图 1-23 所示。Teichmuller 映射和曲面的叶状结构具有深刻的联系，而曲面的叶状结构和曲面上所谓的全纯二次微分等价。存在源曲面和目标曲面的叶状结构，Teichmuller 映射将每条叶子映射成叶子，奇异点映射成奇异点。目前，依然没有成熟算法找出 Teichmuller 映射对应的叶状结构。

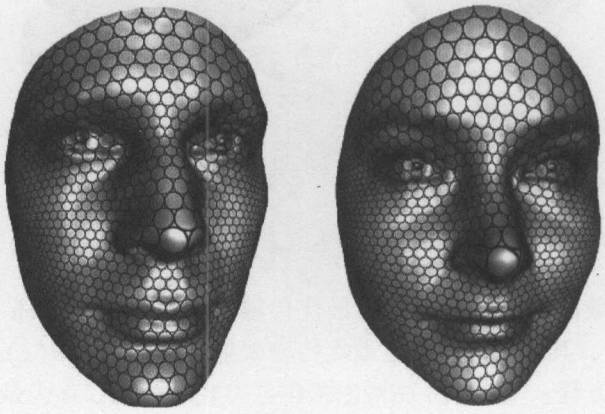


图 1-23　Teichmuller 映射

6. 映射的极分解

任意给一个平面区域间的微分同胚 φ，都唯一地存在分解方式 $\varphi = \nabla u \circ s$，这里 s 是保面积映射，∇u 是最优传输映射，这被称为映射的极分解。如图 1-24 所示，左帧和中帧的映射是保面积映射，中帧和右帧是最优传输映射。

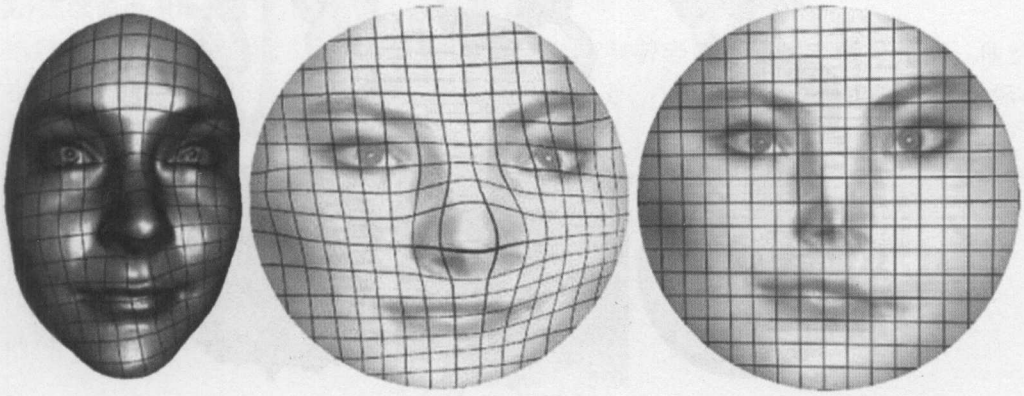


图 1-24　映射的极分解

所有的保面积映射构成一个无穷维的李群，保体积微分同胚群 SDiff，可以由不可压缩流体理论来计算。例如，不可压缩流体的流场必然是散度为 0 的矢量场，不可压缩流体的流动诱导了保面积的映射。SDiff 中可以被配上黎曼度量，从而给出测地线。保体积微分同胚群内有一个自然的黎曼度量：我们在保体积微分同胚群内构造一条路径

$$\{\varphi_t, t \in [0,1]\} \subset \mathrm{SDiff}(\Omega, \mu)$$

连接着两个同胚 $\varphi_0 = id$，$\varphi_1 = \varphi$。这条路径的长度可以计算

$$l\{\varphi(t,\cdot)\} := \int_0^1 \left(\int_\Omega |\partial_t \varphi_t(p)|^2 d\mu(p) \right)^{1/2} \mathrm{d}t$$

两个保体积微分同胚之间的距离定义为连接它们的所有路径长度中最短者。这一理论在医学图像领域被广泛应用于研究曲面的形变。

7. 小结

曲面映射理论非常丰富，所用的数学工具也比较现代而庞杂，包括代数拓扑、微分拓扑、几何偏微分方程、计算几何、流体力学等很多分支。许多关键的理论依然只停留在理论层面，没有相应的实际算法，这为年轻学生提供了广阔的探索天地。我们相信，这些理论必将为很多工程应用提供强有力的支撑，为下一代科技的突破指明方向。

1.5　算法和理论（V）

如图 1-25 所示，曲面单值化定理：所有带度量的封闭曲面都可以保角地映射到三种标准空间中的一种：球面、欧氏平面、双曲平面。

计算机图形学中的曲面参数化（Surface Parameterization）研究长期追求的基本目标之一是保角参数化（Angle - preserving Parameterization）和保面元参数化（Area - preserving Parameterization）。幸运的是，现代几何为这两种参数化技术提供了坚实的理论基础。本节我

们详细介绍保角参数化的理论和算法，为了这一目的，我们将经典的曲面里奇流方法推广到离散情形，建立了离散里奇流理论。

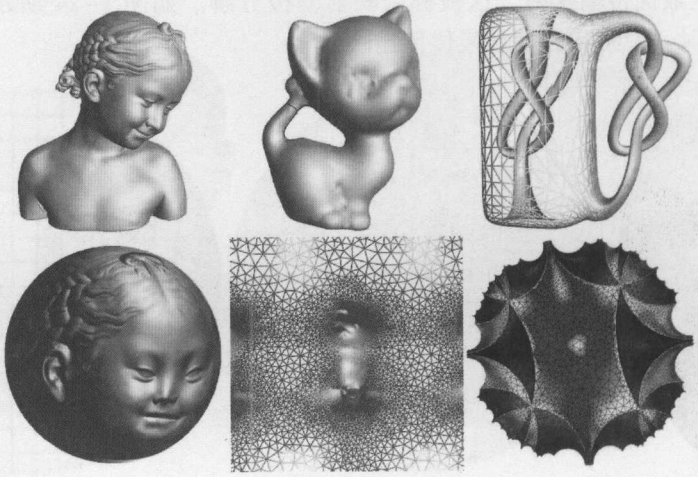


图 1-25　曲面单值化

现存的保角参数化方法有很多种，不同的方法适用于不同的拓扑，解决不同的应用问题，基于不同的数学理论，具有不同的理论深度、工程难度和计算复杂度。这些方法各有千秋，彼此无法相互取代。我们在这里介绍最为普适的一种方法，适用于任何拓扑和任何目标曲率：曲面**里奇流**方法（Surface Ricci Flow）。

里奇流方法是由丘成桐先生创立的几何分析学派的经典方法，由丘先生的好朋友和长期合作者哈密尔顿（Richard Hamilton）提出，用于解决**庞加莱猜想**。庞加莱猜想是说如果在一个封闭的有限拓扑空间中，所有的圈都可以缩成一个点，则这个空间和球面拓扑等价。哈密尔顿的直观想法如下：首先我们在这个拓扑流形上任选一个黎曼度量，然后我们将这个度量进行形变，形变的速率和当前的曲率成正比，那么曲率将遵循非线性热流的规律进行扩散，直至成为常值曲率，那么最终的常曲率度量就是球面的黎曼度量，因此流形和球面等距，从而庞加莱猜想得到证实。这种方法实际上给出了一个非常强有力的工具：**根据目标曲率来设计目标黎曼度量**。这种工具在实际应用中所起到的作用无论如何评价都不会过分。

但从经典的 Ricci 流理论到实用的算法之间有一条难以逾越的鸿沟。经典的 Ricci 流理论是建立在光滑流形上面的，如曲率的定义要求流形的黎曼度量张量是至少二阶可微的。但在计算机中绝大多数几何曲面的表示是欧氏空间中的多面体（Polyhedron），多面体的黎曼度量并不光滑，经典的曲率无法直接在多面体上定义。我们也将多面体曲面称为离散曲面。光滑结构的欠缺使得经典的黎曼几何理论无法直接转化成计算机上的算法。

那么，一个深刻的问题在于：经典微分几何理论中对于流形光滑结构的要求是本质的，还是人为的呢？陈省身先生年轻时曾经给出了微分几何中高斯 - 博纳定理（Gauss - Bonnet Theorem）的内蕴证明，从而一鸣惊人，奠定其微分几何大师的地位。高斯 - 博纳定理说曲面上的黎曼度量无论如何选取，其高斯总曲率是一个拓扑不变量，和具体度量的选取无关。他也曾经说过："高斯 - 博纳定理是否真正需要光滑结构，这是一个值得商榷的问题。"陈先生的真正意思是**许多深刻而本质的几何真理，实际上和流形是否可微无关，它们在离散流形**

上依然成立。高斯－博纳定理和 Ricci 流理论在离散曲面上就是如此，并且离散理论的“极限”就是光滑理论。

下面，我们先简介经典的光滑曲面上的 Ricci 流理论，然后介绍我们自己创立的离散曲面 Ricci 流理论。

1. 连续曲面 Ricci 流

给定一个光滑封闭曲面 S，配有黎曼度量 $\boldsymbol{g}$。陈省身曾经教过杨振宁微分几何，他给杨振宁出过一道题：对于任意一点 $p \in S$，存在一个邻域 $U(p)$，在此邻域上存在**等温坐标**(x, y)，使得黎曼度量具有形式：

$$\boldsymbol{g} = e^{2\lambda(x,y)}(\mathrm{d}x^2 + \mathrm{d}y^2)$$

在等温坐标下，高斯曲率具有非常简洁的形式：

$$K(x,y) = -\Delta_g \lambda(x,y) = -e^{-2\lambda(x,y)}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\lambda(x,y)$$

高斯－博纳定理说总高斯曲率是拓扑不变量：

$$\int_S K\mathrm{d}A = 2\pi\chi(S)$$

这里 $\chi(S)$ 是曲面的欧拉示性数（Euler Characteristic number）。

保角映射的另一种等价提法就是变换曲面的度量，如果映射是保角的，那么新旧度量之间相差一个标量函数，$\bar{\boldsymbol{g}} = e^{2\mu}\boldsymbol{g}$，这里函数 $\mu: S \to \mathbb{R}$ 称为共形因子。那么新的黎曼度量诱导的高斯曲率满足 Yamabe 方程

$$\bar{K} = e^{-2\mu}(-\Delta_g \mu + K)$$

边界的测地曲率也满足类似的方程。反过来，如果我们给定目标曲率 $\bar{K}$，我们可以反解出共形因子 μ，从而得到相应的黎曼度量 $\bar{\boldsymbol{g}}$。如图 1-26 所示，从人脸曲面到平面圆盘的保角变换称为黎曼映照。寻找黎曼映照等价于寻找一个度量 $\bar{\boldsymbol{g}}$，和初始度量相差一个函数，使得高斯曲率 $\bar{K}$ 处处为 0，边界侧地曲率为常数。因此，共形变换等价于求解 Yamabe 方程。

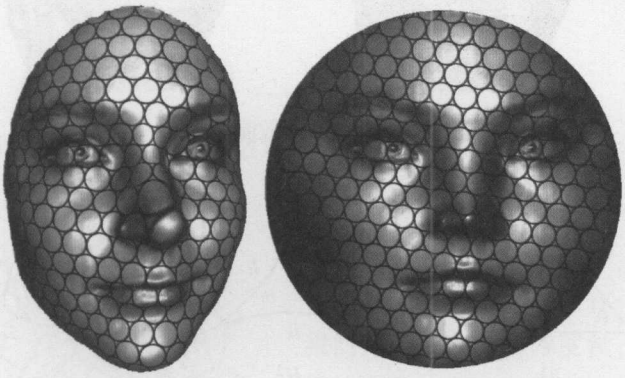


图 1-26　黎曼映照

但是，Yamabe 方程高度非线性，传统的偏微分方程求解方法对其无能为力。最为有效的是哈密尔顿（Richard Hamilton）发明的 Ricci 流方法：

$$\frac{\mathrm{d}g_{ij}}{\mathrm{d}t} = \left(\frac{2\pi\chi(S)}{A(0)} - 2K\right)g_{ij}$$

这里度量张量 $\boldsymbol{g}=(g_{ij})$，$A(0)$是曲面初始的总面积。如果我们用曲率来表示 Ricci 流，则 $\mathrm{d}g_{ij}/\mathrm{d}t=-2Kg_{ij}$对应的曲率常微分方程为：

$$\frac{\mathrm{d}K}{\mathrm{d}t}=\Delta_{\mathrm{g}}K+2K^2$$

这是一个反应－扩散方程。哈密尔顿证明，如果曲面的欧拉示性数非正，则 Ricci 流收敛到常值高斯曲率；Ben Chow 证明，如果曲面的欧拉示性数为正，则 Ricci 流也收敛到常值高斯曲率。这给出了经典的曲面单值化定理：所有的封闭带度量曲面都可以保角地映射到三种标准曲面中的一种：球面、欧氏、双曲曲面，如图 1-25 所示。

佩雷尔曼（Perelman）证明，哈密尔顿的 Ricci 流实际上是所谓熵（Entropy）能量的梯度流，对于欧拉示性数为负的曲面，熵能量为凸能量，常值曲率度量为熵能量唯一的极值点；对于欧拉示性数为零的曲面，情形类似；对于欧拉示性数为正的曲面，熵能量非凸，常值曲率度量为熵能量的鞍点。这预示着直接求球面度量具有本质的困难。

2. 离散曲面 Ricci 流

在计算机上，绝大多数曲面被表示成离散形式，如图 1-27 所示。所谓的**离散曲面**，就是将许多欧氏三角形，沿着边界等距地粘贴在一起，形成一个二维流形。在工程领域，离散曲面被表示成其三角剖分，被称为三角网格 $M(V,E,F)$，这里 V,E,F 表示顶点、边和面的集合。（我们也可以将双曲三角形，或者球面三角形沿着边界粘贴在一起，如图 1-28 所示，构成离散曲面。）

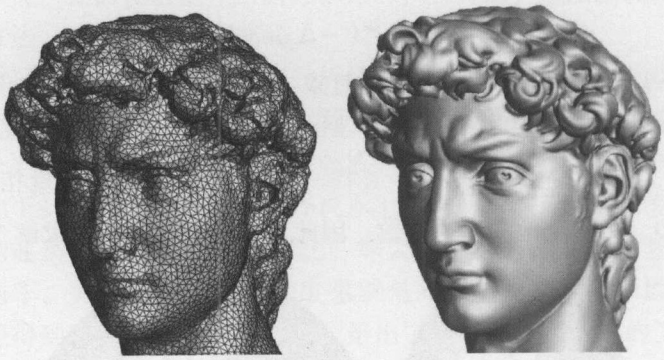


图 1-27　离散曲面

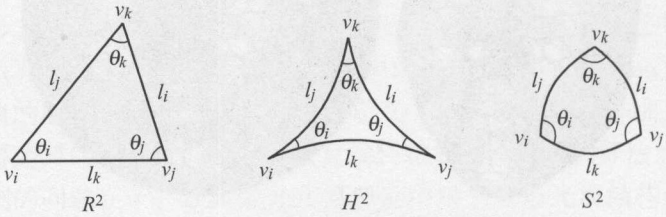


图 1-28　常曲率三角形

每个三角形的几何完全由其边长所决定，因此**离散黎曼度量**就是边长。换言之，离散度量就是定义在边长上的正值函数 $l:E\to\mathbb{R}^+$，在每一个三角形上边长满足三角形不等式。如图 1-28 所示，常曲率三角形的边长满足：

$$l_i + l_j > l_k$$

固定一个离散曲面的组合结构，则其上所有可能的离散度量构成一个凸集，如图 1-29 所示。

离散曲率被定义成角欠（Angle Deficit），对于内顶点，角欠就是围绕顶点的周角和 2π 的差别；对于边界顶点，角欠就是围绕顶点的周角和 π 的差别。

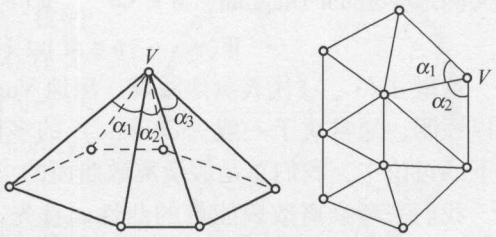


图 1-29　离散曲率

通过简单的组合推理，我们可以轻易证明**离散高斯 - 博纳定理**：总曲率等于 2π 和欧拉示性数的乘积

$$\sum_{v \notin \partial M} K(v) + \sum_{v \in \partial M} K(v) = 2\pi\chi(M)$$

更为深刻地，我们可以证明给定光滑曲面，我们可以在曲面上稠密采样，计算采样点的测地 Delaunay 三角剖分，然后用欧氏三角形来取代测地三角形，这样得到的离散曲面可以逼近光滑曲面，离散曲率逼近光滑曲率测度。由此，我们可以用离散高斯 - 博纳定理来证明光滑高斯 - 博纳定理。这直接验证了陈省身先生的断言。

在连续情形下，黎曼度量决定高斯曲率；在离散情形下，边长决定三角形内角，这就是通常的余弦定理：

$$\cos\theta_k = \frac{l_i^2 + l_j^2 - l_k^2}{2 l_i l_j}$$

关键的概念是度量的**离散共形变换**。我们在顶点上定义**离散共形因子** $u: V \to \mathbb{R}$，给定边 $[v_i, v_j]$，其边长变换满足如下规律：

$$l_{ij} \to e^{u_i} l_{ij} e^{u_j}$$

这一操作被称为**顶点缩放操作**（Vertex Scaling）。

给定目标曲率 $\bar{K}: V \to \mathbb{R}$，满足高斯 - 博纳条件，那么**离散曲面 Ricci 流**和连续 Ricii 流的定义方式相同：

$$\frac{\mathrm{d}u(v)}{\mathrm{d}t} = 2(\bar{K}(v) - K(v))$$

佩雷尔曼所发现的熵能量，也有离散对应形式。**离散熵能量**定义为：

$$E(\mathbf{u}) = \int^{\mathbf{u}} \sum_i (\bar{K}_i - K_i)\,\mathrm{d}u_i$$

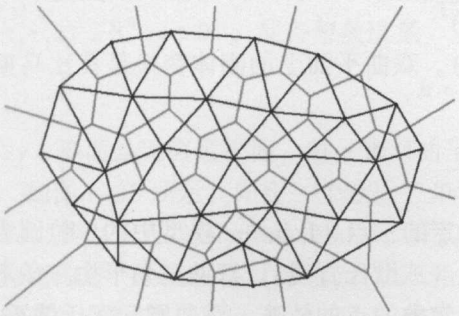


图 1-30　Delaunay 三角剖分，对偶的 Voronoi Diagram

因此，在实际计算中，我们可以用牛顿法直接优化离散熵能量。在优化过程中，每一步我们得到一个多面体度量，我们将三角剖分不停地更新，使之在当前度量下是 Delaunay 三角剖分。换言之，**在曲面离散 Ricci 流算法中，三角剖分一直保持 Delaunay**。

3. 离散 Ricci 流解的唯一性

如图 1-30 所示，我们可以在离散曲面上

定义测地 Voronoi Diagram，$M=\bigcup_{v_i\in V}W(v_i)$，这里 Voronoi 胞腔

$$W(v_i)=\{p\in M\mid d_g(v_i,p)\leqslant d_g(v_j,p),\forall v_j\in V\}$$

这里 $d_g(\cdot,\cdot)$代表测地距离。测地 Voronoi Diagram 的对偶称为测地 Delaunay 三角剖分。可以证明，通常对于一般（Generic）的多面体度量，Delaunay 三角剖分存在并且唯一。在以下的讨论中，我们总是假设离散曲面的三角剖分是 Delaunay 的。

我们来考察离散熵能量的凸性。首先，每条 Voronoi Diagram 的边，都对应于 Delaunay 三角剖分的一条边 $e\in E$，两条边的比值称为这条 Delaunay 边的权重 $w(e)$。Delaunay 三角剖分保证边的权重非负 $w(e)\geqslant 0$。由此我们定义曲面的**离散 Laplace－Beltrami 算子**，给定任意一个定义在顶点集上的离散函数 $f:V\to\mathbb{R}$，离散函数可以利用重心坐标线性拓展成分片线性函数

$$\Delta f(v_i)=\sum_{[v_i,v_j]\in E}w_{ij}(f(v_j)-f(v_i))$$

通过直接计算我们可以证明

$$\frac{1}{2}\boldsymbol{f}^{\mathrm{T}}\Delta\mathbf{f}=\int_M|\nabla f|^2\mathrm{d}A$$

由此我们证明离散 Laplace－Beltrami 算子是半正定矩阵，其零空间是直线，由向量$(1,1,\cdots,1)^{\mathrm{T}}$生成。离散熵能量的海森矩阵（Hessian matrix）就是离散 Laplace－Beltrami 算子，因此熵能量在线性子空间 $\{\sum_i u_i=0\}$ 上是严格凸的。

离散熵能量的梯度是目标曲率和当前曲率之差，$\nabla E(\boldsymbol{u})=\overline{K}(\boldsymbol{u})-K(\boldsymbol{u})$。如果共形因子的定义域是一个凸集，那么由熵能量在空间 $\{\sum_i u_i=0\}$ 上的凸性，我们得到映射 $\boldsymbol{u}\to E(\boldsymbol{u})$ 是微分同胚（光滑双射）。这意味着**共形因子到曲率的映射是微分同胚**，即离散 Ricci 流的解的唯一性得以保证。

那么，离散共形因子的定义域是否为凸集呢？这依赖于离散曲面 Ricci 流解的存在性证明。存在性证明正是离散 Ricci 流理论最为困难的部分。为此，我们先简介一些双曲几何的基本理论。

4. 平面双曲几何

复平面单位圆盘$\mathbb{D}^2:=\{z\in\mathbb{C}\mid|z|<1\}$上配备黎曼度量

$$\boldsymbol{h}=\frac{4\mathrm{d}z\mathrm{d}\bar{z}}{(1-z\bar{z})^2}$$

则所得为双曲平面的庞加莱模型（Poincaré s Disk）。双曲平面上的刚体变换是莫比乌斯变换，例如

$$z\to e^{i\theta}\frac{z-z_0}{1-\bar{z}_0z}$$

莫比乌斯变换可以将边界上的任意三点变成指定的三点。Poincare 模型中的单位圆为无穷远点，双曲直线是和单位圆相互垂直的欧氏圆弧（或欧氏直线）。双曲三角形由三条双曲直线围成的多边形；双曲圆是到双曲圆心距离等于常数的点的轨迹，双曲圆和欧氏圆重合，但是圆心不重合。

我们主要应用一种特殊的双曲三角形，ideal hyperbolic triangle，其三个顶点都在无穷

远处，边长为无穷大，三个内角为0，但是面积为π。所有的 ideal hyperbolic triangles 彼此等距。同时，我们主要考虑一种特殊的双曲圆，horocycle，其圆心在无穷远处，半径也是无穷大。如图1-31所示，Farey - Ford 镶嵌模式就是由 ideal hyperbolic triangles 和 horocycles 构成的。

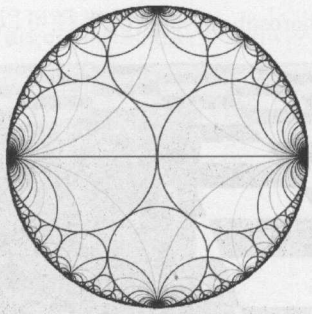


图1-31　Ideal hyperbolic triangles and Horocycles（Farey - Ford tessellation by Richard Green）

给定一个 ideal triangle，我们用三个 horocyles 来截除三个顶点，得到一个 decorated triangle，如图1-32所示。我们用红边来表示 decorated triangle 的三条边，其有向双曲边长记为$\{l_i,l_j,l_k\}$。如果两个 horocycles 彼此相交，则边长为负，如图1-32右图所示；蓝边表示三个 horocycle 的圆弧，其双曲长度记为$\{\alpha_i,\alpha_j,\alpha_k\}$，我们可以称之为三个内角。给定任意的三个实数$\{l_i,l_j,l_k\}$，存在唯一的 decorated triangle，其边长是$\{l_i,l_j,l_k\}$。Decorated triangle 也满足特定的余弦定理（Cosine Law）：

$$\alpha_i=\frac{L_i}{L_jL_k},\quad L_i=e^{l_i/2}$$

这里$\{L_i,L_j,L_k\}$称为 Penner 的 λ - 长度。

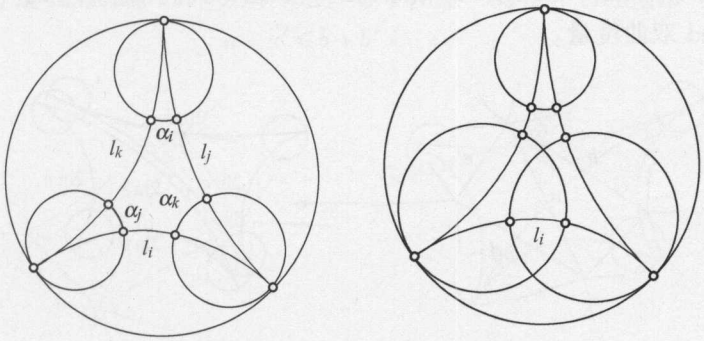


图1-32　Penner s decorated triangle

5. 曲面双曲几何

下面我们讨论如何构建曲面上的双曲度量。我们考虑三维双曲空间的庞加莱模型，$\{(x,y,z)\in\mathbb{R}^3\mid z>0\}$，配备黎曼度量

$$\boldsymbol{h}=\frac{\mathrm{d}x^2+\mathrm{d}y^2+\mathrm{d}z^2}{z^2}$$

则 xy - 平面是无穷远平面，每一张双曲平面都是赤道在无穷远平面上的半球面。

如图1-33所示，给定一个欧氏三角形，具有边长$\{l_i,l_j,l_k\}$，我们构造一个 decorated triangle，其边长为$\{2\ln l_i,2\ln l_j,2\ln l_k\}$。我们在无穷远平面上画一个欧氏三角形，然后以此三角形为底构造一个三棱柱（绿色），以三角形外接圆为赤道做一个半球（蓝色），则三棱柱和半球面的交集是一个 ideal triangle。然后，我们构造三个 horospheres（红色、黄色、粉色），它们和无穷远平面相切在三角形的三个顶点，horospheres 将 ideal triangle 三个顶点截除，得到 decorated triangle，如图1-34所示，蓝色的曲面代表了 decorated triangle。在这个过程中，

三个 horosphere 的欧氏半径可以由$\{l_i,l_j,l_k\}$计算出来。

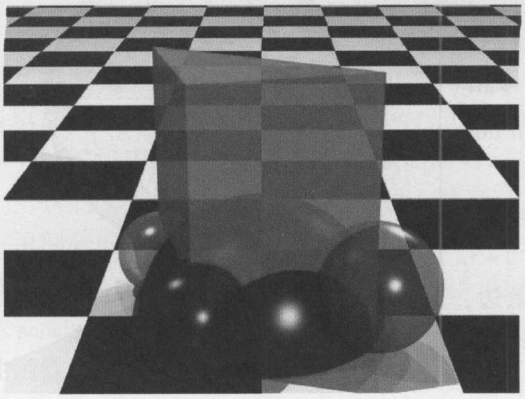


图 1-33　Decorated hyperbolic triangle

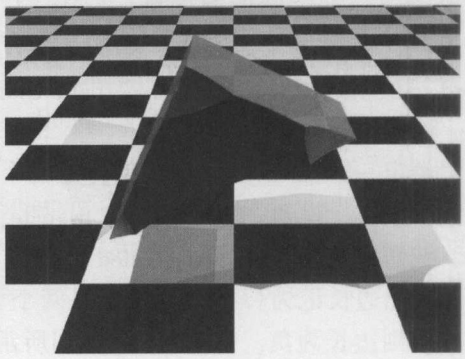


图 1-34　decorated triangle（蓝色）

给定封闭拓扑曲面 S，和曲面上的有限离散点集 V，记为$\{S,V\}$。我们在(S,V)配备一个欧氏度量，使得所有的离散曲率集中在顶点 V 上，则在通常情况下存在唯一的 Delaunay 三角剖分 T。对于每条边，我们将相邻的两个欧氏三角形同时转换成 decorated 双曲度量，将欧氏长度变换成 Penner 的 λ - 长度，$(l_i,l_j,l_k)\rightarrow(2\ln l_i,2\ln l_j,2\ln l_k)$，如图 1-35 所示，这种转换方式给出了两个 decorated triangles 的粘贴方式。如此，我们将(S,V)的一个欧氏度量转换成了一个 decorated 双曲度量。

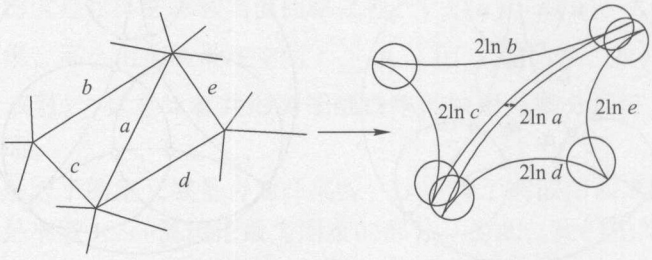


图 1-35　欧氏度量到 decorated hyperbolic 度量的转换

如果我们忽略 horocycles，那么每一个欧氏三角形都被转换成一个 ideal 双曲三角形，由图 1-36 所示的粘贴方式，这些 ideal 双曲三角形黏在一起，构成了(S,V)上的一个双曲度量，每个顶点都成为一个无穷远的尖点（Cusp）。我们将这个度量称为(S,V)的一个带尖点的双曲度量（Hyperbolic Metric with Cusp），记为 $\boldsymbol{h}$。然后，我们用 horosphere 来截除顶点处的尖点，得到 decorated hyperbolic metric，记为$\widetilde{\boldsymbol{h}}$。在顶点 $v_i\in V$ 处，horosphere 和曲面$(S,V,$

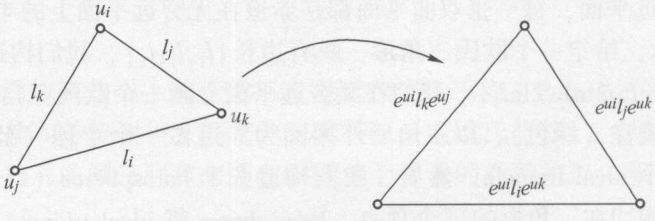


图 1-36　欧氏度量的顶点缩放操作

$\boldsymbol{h}$)交线的双曲长度记为 w_i，称次顶点处的 decoration。那么 decorated metric $\widetilde{\boldsymbol{h}}$由 hyperbolic metric $\boldsymbol{h}$，和 decorations $(w_1, w_2, \cdots, w_n)$共同决定。

从欧氏度量到 decorated 双曲度量的变换具有如下的关键性质：将图 1-37 所示的欧氏度量的顶点缩放操作，变换成 decorated 双曲度量 decoration 缩放操作，即 horosphere 的缩放操作。

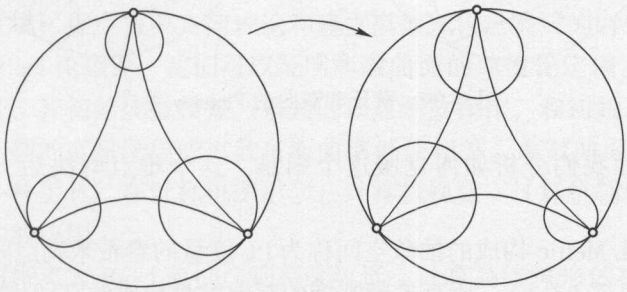


图 1-37　decorated 双曲度量的 decoration 缩放操作

6. Delaunay 三角剖分

我们先来定义 Delaunay 三角剖分。如图 1-38 所示，欧氏度量下，decorated 双曲度量下，一个三角剖分是 Delaunay 的冲要条件是：对于任意一对相邻的三角形

$$a + a' \leqslant b + b' + c + c'$$

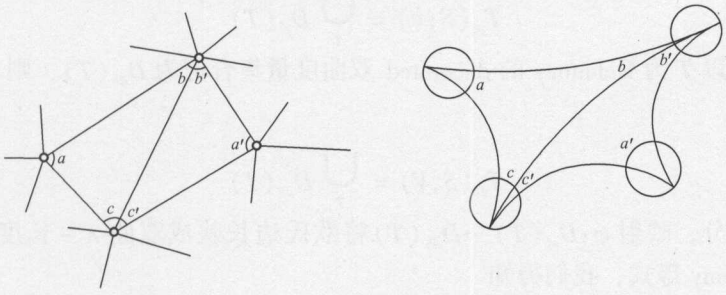


图 1-38　欧氏和双曲 Delaunay 三角剖分

我们可以直接证明，图 1-35 所示的从欧氏度量到 decorated hyperbolic 度量的转换将欧氏 Delaunay 三角剖分映成双曲 Delaunay 三角剖分。

在特殊情况下，同样的欧氏度量，会有两个 Delaunay 三角剖分，这时存在两个相邻的三角形，四个顶点共圆，如图 1-39 左图所示。这时，边长满足所谓的 Ptolemy 等式：

$$CC' = AA' + BB'$$

这两个 Delaunay 三角剖分相差一个对角线对换。那么，欧氏度量对应的双曲度量的 Delaunay 三角剖分也不唯一，彼此也相差一个对角线对换，如图 1-39 右图所示，双曲的 λ - 长度也满足 Ptolemy 等式。

7. 离散 Ricci 流解的存在性

离散 Ricci 流的过程中，离散曲面的黎曼度量光滑变化，但三角剖分不停地在改变。通常的理论讨论中，都假设离散曲面的三角剖分保持一致，因此**动态三角剖分**是理论分析中最

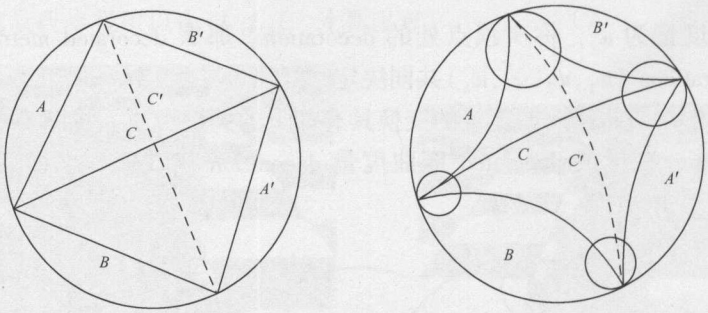


图 1-39　欧氏和双曲的 Ptolemy 等式

为困难的部分。下面我们分析如何克服这个困难，这个难点解决后，其他部分就会迎刃而解。

(S,V)上所有 PL Metric 构成的抽象空间称为 PL 度量的泰希米勒空间（Teichmuller Space of PL Metrics），记为 $T_{pl}(S,V)$；所有的带装饰的双曲度量构成的空间称为带装饰的双曲度量泰希米勒空间（Teichmuller Space of Decorated Hyperbolic Metrics），记为 $T_D(S,V)$。我们的**目的在于建立两个泰希米勒空间之间的同胚**。图 1-35 已经给出了这个映射：**在 Delaunay 三角剖分下，欧氏边长映成双曲的 λ - 长度**。

我们固定(S,V)的一个三角剖分 T，考察所有以 T 为 Delaunay 的 PL 度量，其集合记为 $D_{pl}(T)$，可以证明 $D_{pl}(T)$是一个单联通的集合，由此我们得到 $T_{pl}(S,V)$的一个胞腔分解：

$$T_{pl}(S,V) = \bigcup_T D_{pl}(T)$$

同样，所有以 T 为 Delaunay 的 decorated 双曲度量集合记为 $D_{dh}(T)$，则 $T_D(S,V)$的胞腔分解为：

$$T_D(S,V) = \bigcup_T D_{dh}(T)$$

固定三角剖分，映射 $\varphi: D_{pl}(T) \to D_{dh}(T)$将欧氏边长映成双曲 λ - 长度，为微分同胚。同时，由于 Ptolemy 等式，我们得知

$$\varphi: D_{pl}(T) \cap D_{pl}(T') \to D_{dh}(T) \cap D_{dh}(T')$$

也是同胚。因此，φ 在每个胞腔内都是微分同胚，在胞腔交界处是同胚，整体上 φ 是两个泰希米勒空间之间的拓扑同胚。

直观而言，我们固定一个 PL 度量(S,V)，就固定了(S,V)的一个带尖点的双曲度量 $\boldsymbol{h}$，我们对 PL 度量进行顶点缩放操作，等价于对双曲度量进行 decoration 缩放变换。我们更新 PL 度量下的 Delaulany 三角剖分，双曲度量 $\boldsymbol{h}$ 和 decoration 并不变化。由此，我们去除了动态三角剖分带来的困难。

由此，我们构造复合映射：

$$F: \boldsymbol{h} \times R^n \to T_D(S,v) \to T_{pl}(S,V) \to (-\infty, 2\pi)^n \cap \left\{ \sum x_i = 2\pi\chi(S) \right\}$$

这里输入是顶点上的 decorations，第一个箭头是包含映射，第二个箭头是 φ^{-1}，第三个箭头是离散高斯曲率。那么映射 F 限制在$\{\pi x_i = 1\}$为同胚。由此，我们再经过简单的推理，即可证明曲面离散 Ricci 流解的存在性。

1.6　总结

回顾以上证明过程，我们看到需要一些比较深入的双曲几何和 Teichmuller 空间理论。虽然结论和算法貌似初等，但其后面的理论基础却是非常现代而深奥的，并且和连续理论所用的数学工具迥然不同。这显示了将连续理论推广成离散理论的内在难度。但是，为了适应计算机技术迅猛的发展，推广古典几何理论和建立离散理论已经成为时代的必然。

根据离散曲面 Ricci 流理论，我们可以证明离散曲面的单值化定理：我们可以找到一个离散度量和三角剖分，和初始度量相差一系列的顶点缩放操作，和初始三角剖分相差一系列的对角线对换操作，新的度量诱导的离散高斯曲率处处相等。离散曲面 Ricci 流提供了设计黎曼度量强有力的一种工具，在计算机图形学、计算机视觉、计算机辅助设计、网络和医学图像等领域都有深入的应用。

展望未来，如何将离散 Ricci 流推广到高维流形，计算高维流形的标准度量将是我们长期追求的目标，更是年轻人值得为之奋斗的目标！

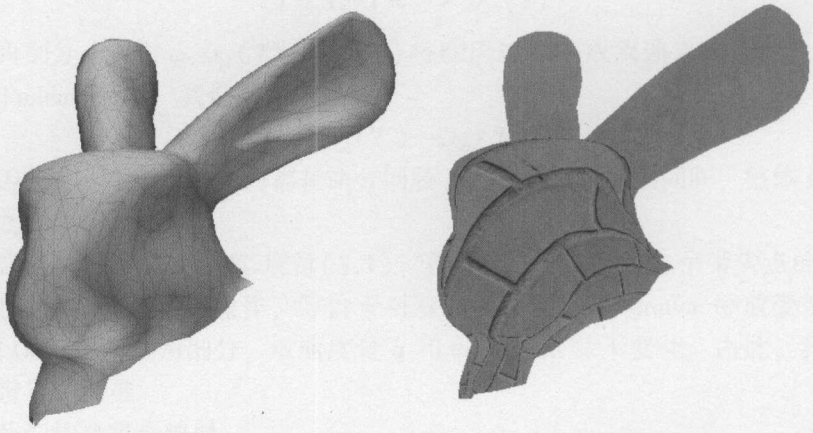
第 2 章 Blender 软件贴图

2.1 贴图介绍

2.1.1 贴图概念

三维模型制作出来后，最终要显示到计算机屏幕上。三维模型的显示和渲染的目标是逼真，也就是具有翔实的细节。通常有两个方法可以逼真地显示和渲染三维模型：第一种方法是在制作时就制作很精致的、具有细节的高精度模型；第二种方法是制作低精度的模型，但在低精度模型上采用贴图的方法。第一种方法制作过程比较复杂，制作出来的模型顶点很多，占用的内存和空间比较大，在运行时速度就比较慢。第二种方法虽然模型本身很简单，不具有细节，但细节包含在贴图里面。这样可以很快速地进行加载和显示。而且用贴图的方法，可以对于同一个三维模型显示出不同的最终效果。在三维游戏中，影视特效里面大部分都是采用贴图的方法来使三维模型更加逼真的。

如图 2-1 所示没有贴图的三维模型和有贴图的三维模型。从中可以看出，有贴图的三维模型显示的效果更佳，具有更多的细节。



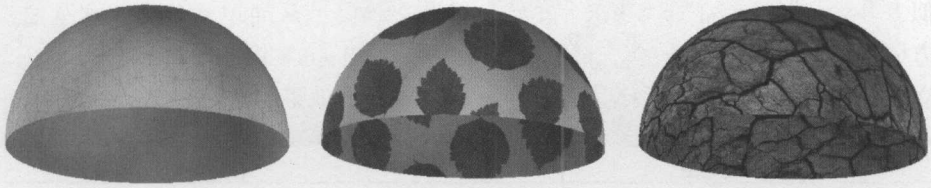
（a）没有贴图的三维模型　　（b）贴图的三维模型

图 2-1　三维模型贴图

贴图是计算机图形学中的一个核心技术，是把存储在内存里的二维图像图包裹到 3D 渲染物体的表面。纹理给物体提供了丰富的细节，能够用简单的方式模拟出复杂的外观。贴图

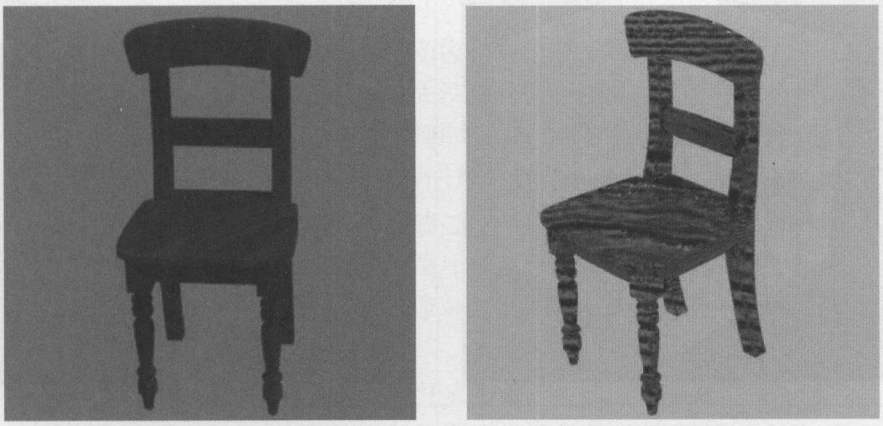
过程就像印花贴到一个平面上一样，可以减少制作三维模型的计算量。例如，可以创建一个球并把脸的纹理贴上去，这样就不用在制作三维模型时制作鼻子和眼睛的形状，只需要在贴图上绘制鼻子和眼睛。最终三维模型显示时就具有鼻子和眼睛。

贴图（Texture）是一张平面的二维图像。贴图的过程就是把二维图像上的像素和三维图像的顶点一一对应。从而在显示三维图像时，可以按照对应的结果显示二维图像相应的像素颜色。同一个三维模型可以对应不同的贴图，不同的贴图能够给人们带来不同的视觉感受，是营造三维模型真实效果的最有效手段之一。也就是即使三维模型是一样的，但由于贴图的不同，最终显示的结果不一样。如图 2-2 所示，同样的半球三维模型，经过不同的贴图后显示的结果不同。贴图的椅子三维模型如图 2-3 所示。

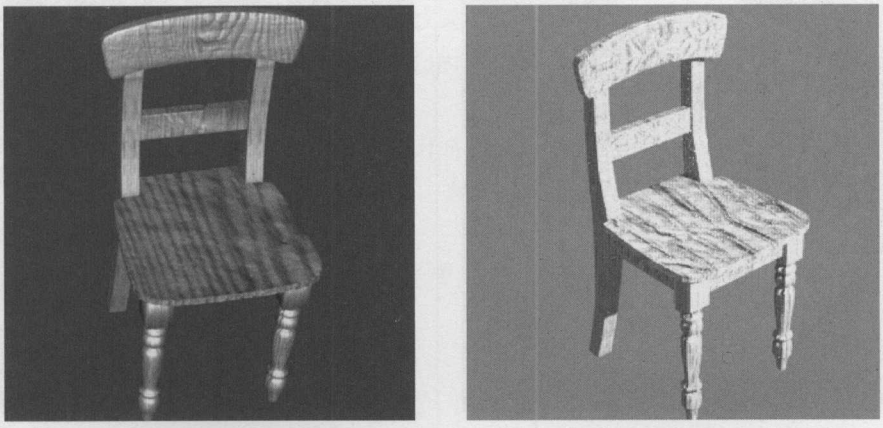


（a）三维模型　　（b）贴图1　　（c）贴图2

图 2-2　不同贴图的三维模型



（a）三维模型　　（b）贴图1



（b）贴图2　　（b）贴图3

图 2-3　贴图的椅子三维模型

在渲染时除了贴图之外，还有一个相关的概念是**材质（Material）**。材质是渲染要用到的各种可视属性的集合。这些属性包含表面的色彩、纹理、光滑度、透明度、反射率、折射率、发光度等。有了这些材质属性，才能够使三维模型显示得更加真实。材质也包含了贴图。除了材质之外，三维模型的最终显示还依赖于光照，根据光照设置的不同，而显示不同的效果。

2.1.2 Blender 软件贴图

本节主要讲述如何为三维模型设置贴图，也就是如何把二维的图像和三维的模型进行对应，这个对应的过程就是贴图的过程。首先介绍 Blender 软件里进行贴图的界面和方法，后续的内容详细讲述贴图算法的原理和实现。所有的三维建模软件都具有贴图的功能，基本方法都类似，我们采用 Blender 软件作为实例。根据三维模型从简单到复杂，本章循序渐进地介绍立方体、球形、兔子、人脸、椅子等三维模型在 Blender 软件中的贴图过程。如图 2-4 所示是立方体、球、兔子三维模型用 Blender 软件贴图的效果。

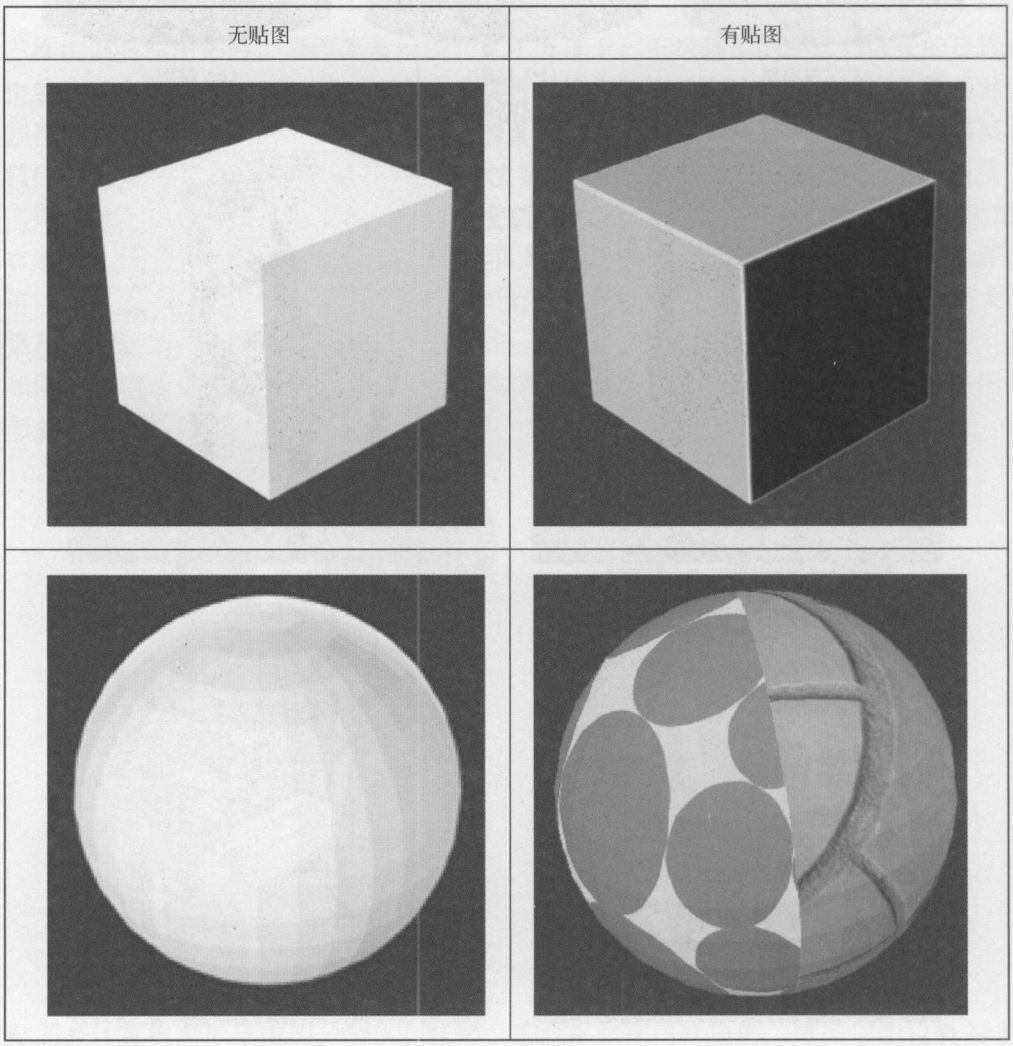


图 2-4　Blender 软件贴图

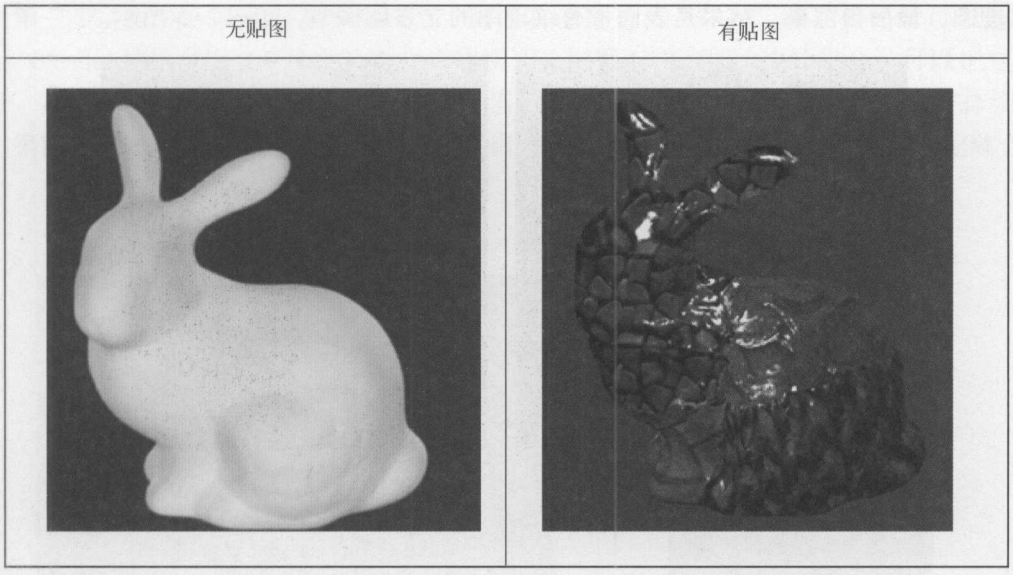


图 2-4　Blender 软件贴图（续）

三维模型的贴图过程分为如下三步：第一步，将封闭的三维模型切开；第二步，把三维模型开成平面；第三步，把展开的平面和二维的图像对应起来。第三步有两种方法使二维图像和展开的平面对应。第一种方法处理二维图像事先没有确定的情况。这时可以根据展开的平面网格形状进行绘制，也就是把展开的平面导出纹理框图像，用图形处理软件在纹理框里涂上颜色或用 Photoshop 等工具在纹理框里加上各种纹理图片。最后将处理好的图片导回 Blender，这样由于二维图像是根据平面绘制的，所以在绘制的过程中就一一对应了。第二种方法是处理图像已经确定的情况。这种方法需要直接导入纹理图片，因为图像是固定不变的，因此需要拉伸调整展开的二维平面网格，使二维平面能够和图像对应。具体步骤如下所述。

1. 绘制图像方法

第一步：展开 UV。将模型切开，然后展开成平面。

第二步：导出和导回 UV。导出纹理框，用图形处理软件（画图等）在纹理框里涂上颜色或用 Photoshop 等工具在纹理框里加上各种纹理图片，再将处理好的图片导回 Blender。

第三步：渲染。调好灯光，调整好摄像机角度，渲染出图。

2. 图像固定方法

第一步：展开 UV。将模型切开，然后展开成平面。

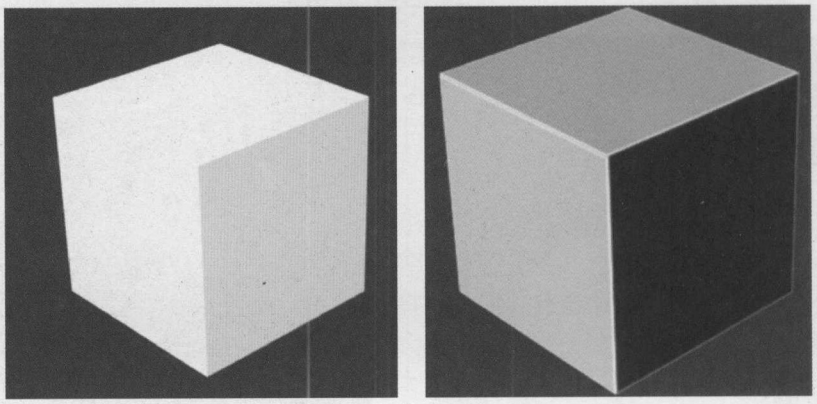
第二步：调整 UV。模型剪开展成平面后，直接导入纹理图片，然后拉伸调整纹理网格使网格正好覆盖纹理图片。

第三步：渲染。调好灯光（一般是三点光源法），调整好摄像机角度，渲染出图。

2.2　立方体贴图

立方体是最简单的三维模型之一，通过立方体贴图，可以了解贴图的方法和使用的界面。立方体贴图将一个立方体模型沿一定顺序的边剪开，展开成平面图形，然后将各个面贴

上纹理图，最后再渲染，结果是表面带有纹理图的立方体模型，如图 2-5 所示。



（a）原始模型　　（b）贴图模型

图 2-5　立方体贴图

1. 立方体贴图步骤

第一步：打开 Blender 软件，Blender 的 3D 视图中默认有一个立方体模型，如图 2-6 所示。

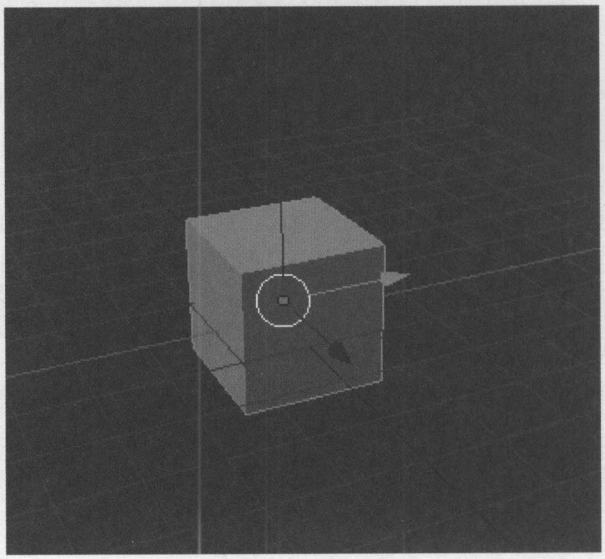


图 2-6　立方体

第二步：将鼠标指针放到立方体上右键单击选中立方体，按“TAB”键进入编辑模式，再选中边编辑模式，如图 2-7 所示。

图 2-7　编辑模式

第三步：选中要标记的边（“Shift”＋鼠标右键选边，“Shift”键不要松开可以连续选边）。Blender 软件会沿着选中的边把立方体切开。根据不同选择的边的集合，可以把立方体切开。如图 2-8 所示其中的一种边的集合。当然也有其他边的集合同样可以将立方体切开。

第四步：按“CTRL＋E”组合键，会弹出如图 2-9 所示的命令框。单击“标记缝合边”命令。

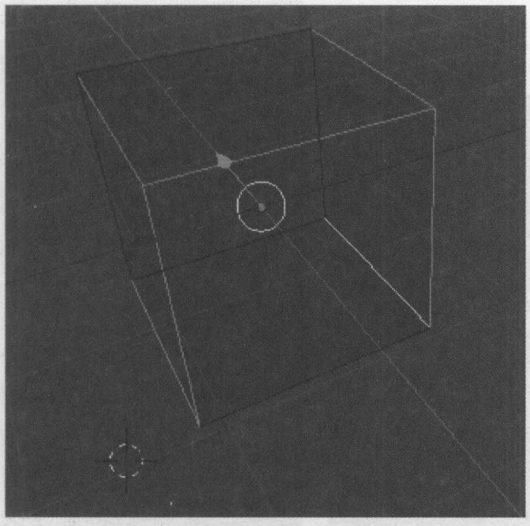


图 2-8　选择边

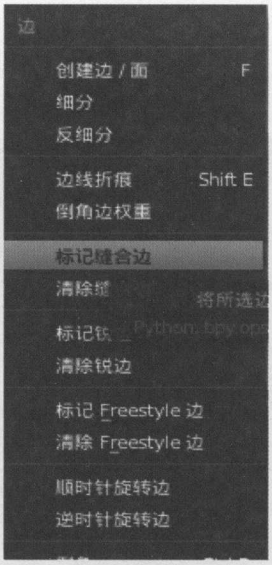


图 2-9　标记缝合边

第五步：按“A”键选中整个立方体，再按“U”键，在弹出的命令框中单击“展开”命令，如图 2-10 所示。这样立方体就成功展开成平面图形了。下一步就能看到展开的图了。

第六步：在另一个窗口的右下角选择“UV/图像编辑器”窗口，如图 2-11 所示。

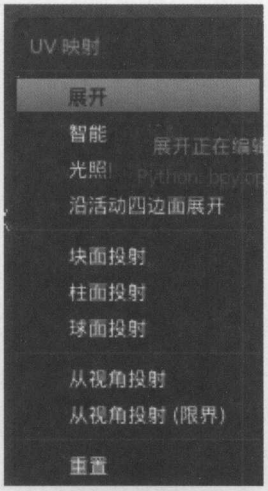


图 2-10　标记缝合边

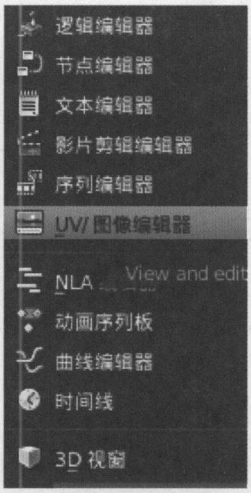


图 2-11　图像编辑器

第七步：从而显示如图 2-12 所示的展开图 。沿不同选择边剪开的展开图肯定会不一样，但这不影响最后的贴图效果。

第八步：单击“UV”里面的“导出 UV 布局图”命令，选择好图片的存储路径然后导出。这一步导出的是一张展开图的框图，如图 2-13 所示。

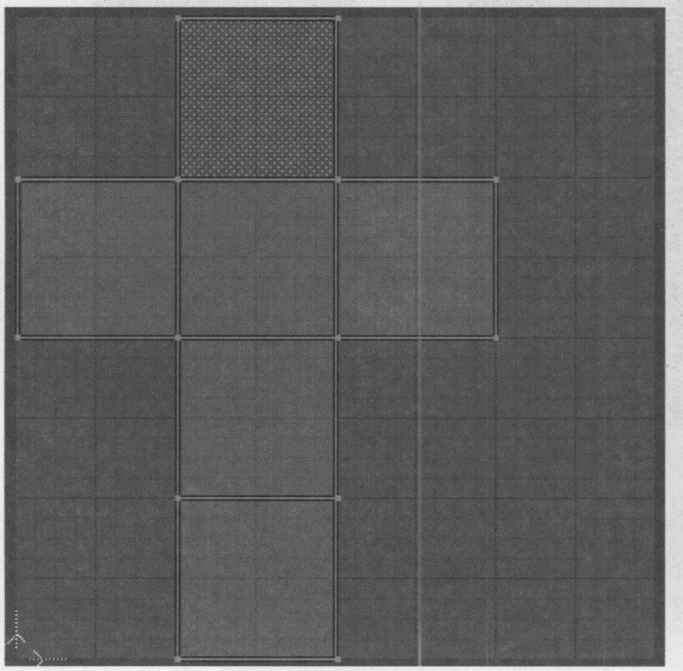


图 2-12　立方体展开图

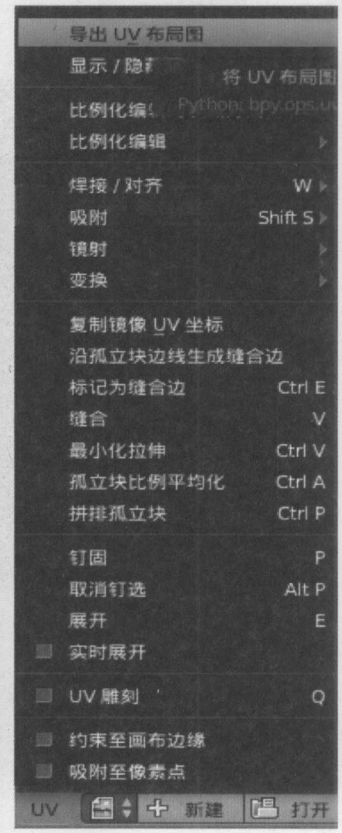


图 2-13　导出布局图

第九步：用画图软件打开导出的图，可以用 Windows 系统自带的“画图”软件，也可以用 Photoshop 或其他图形处理软件。然后在不同的框里涂上不同颜色，也可以贴上其他图片，然后保存。需要注意是，处理时不要改变线框的大小，如图 2-14 所示。

第十步：回到 Blender，在 UV 编辑器的图像菜单中单击“打开图像”命令，打开刚才涂好颜色的图片，然后可以看到线框和图像一一对应了，如图 2-15 所示。

第十一步：进入材质面板和纹理面板，在选项一栏中选中“面纹理”选项，在“映射”一栏的“坐标”中选择“UV”选项，如图 2-16 和图 2-17 所示。

第十二步：打好光照，一般用“三点光源法”。在设置光照时注意要把“高光”前面的钩取消掉，否则渲染出的图会有反光。然后渲染出图。要想得到不同角度的渲染图有两种方法：一是旋转物体；二是调整摄像机位置。

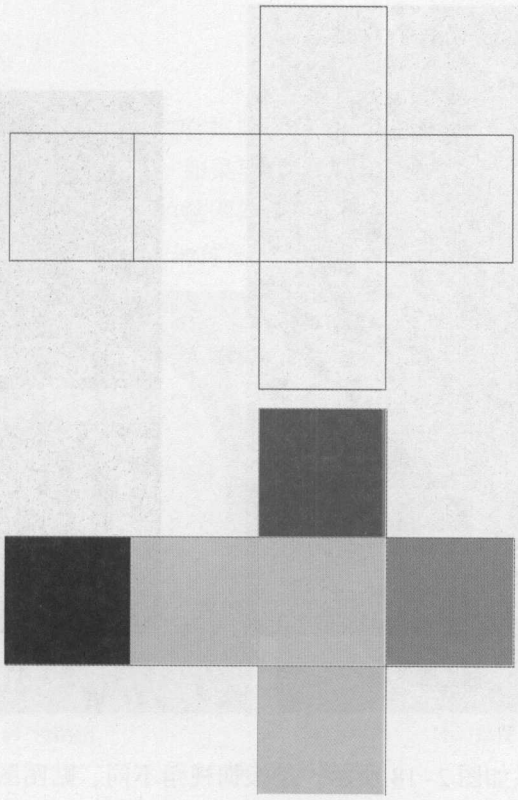


图 2–14　布局图绘制

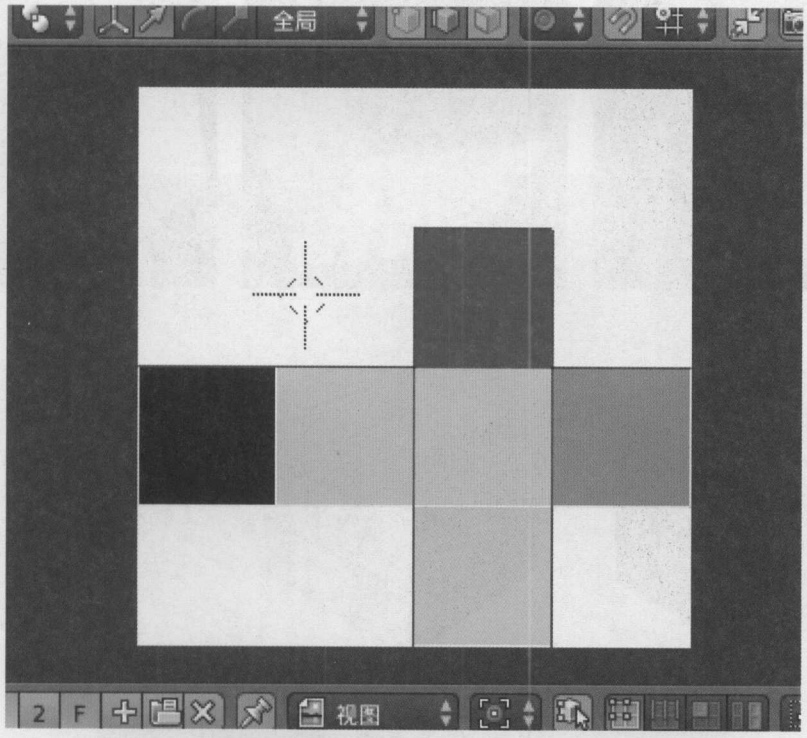


图 2–15　打开布局图

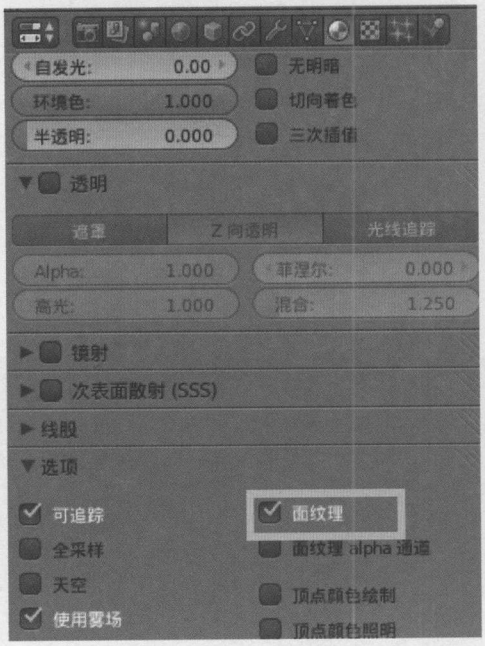


图 2-16　面纹理界面

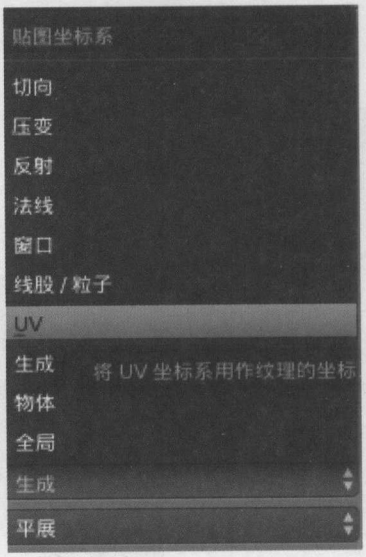
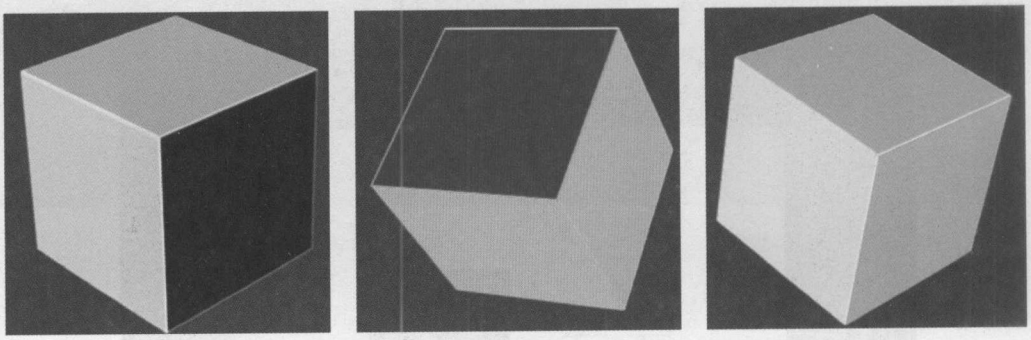


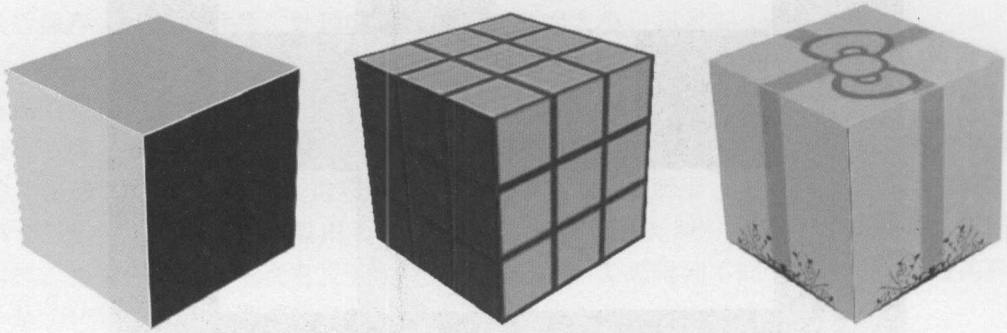
图 2-17　映射界面

2. 效果图

最终贴图的渲染效果如图 2-18 所示，是根据视角不同、贴图图内容不同的显示效果。



（a）视角不同

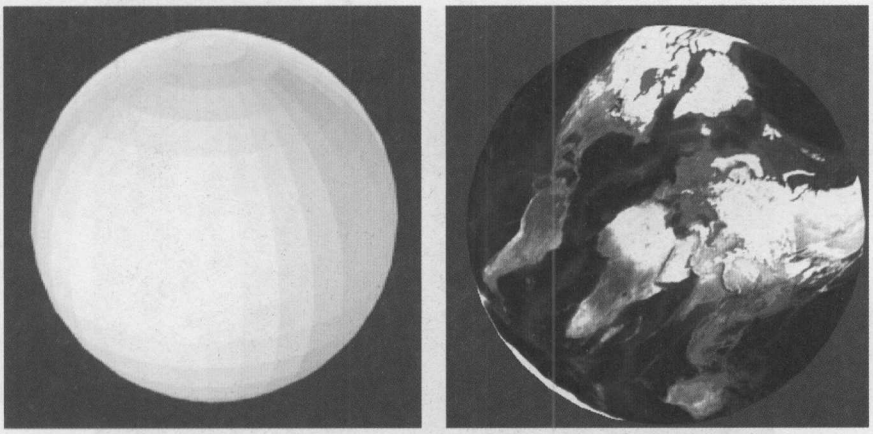


（b）贴图图片不同

图 2-18　贴图的渲染效果

2.3　球形贴图

在立方体贴图中，由于立方体比较有规则，所贴图像也比较简单，因此可以在展开之后再进行图像的绘制。但对于球形来说，如果要显示出地球的效果，就需要先把球形展开，然后再调整展开的平面，使平面和地球的贴图对应上。从而能够采用比较复杂的模型贴图。如图 2-19 所示是球形三维模型贴上一张地球图像的效果。



（a）没贴图　　（b）贴图后

图 2-19　球形贴图

1. 球形贴图步骤

第一步：打开 Blender 软件，删除默认的立方体三维模型，右键单击选中立方体，按“Delete”键删除立方体，或者按“X”键。

第二步：然后添加一个经纬球模型。单击菜单添加—网格—经纬球，添加的球形三维模型如图 2-20 所示。

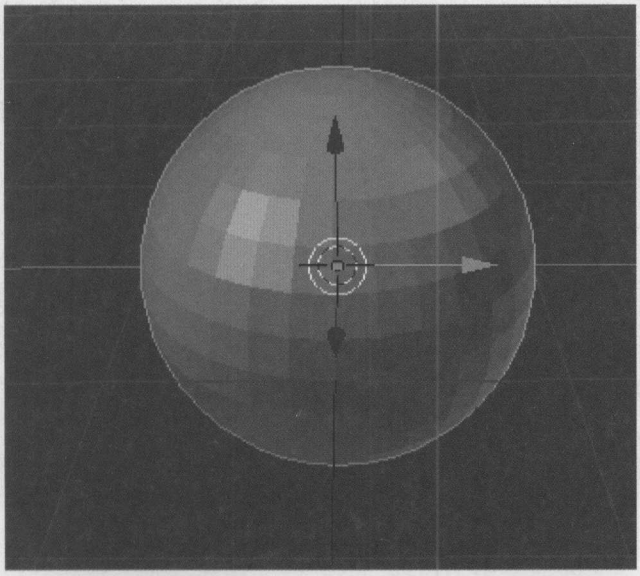


图 2-20　球形三维模型

第三步：右键单击选中这个经纬球，按“Tab”键进入编辑模式，按“A”键全选网格，再按“U”键选择“球面投射”，如图 2-21 所示。

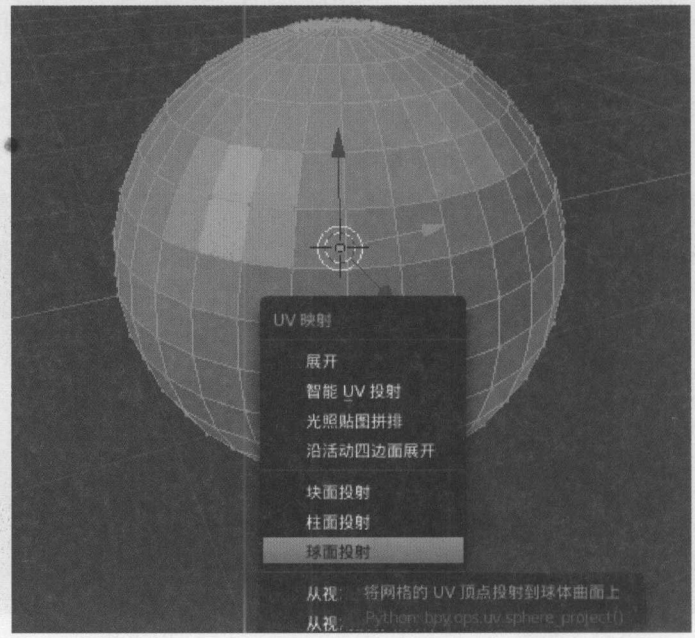


图 2-21　球面投射菜单

第四步：把一个新窗口改成“UV/图像编辑器”窗口，如图 2-22 所示。

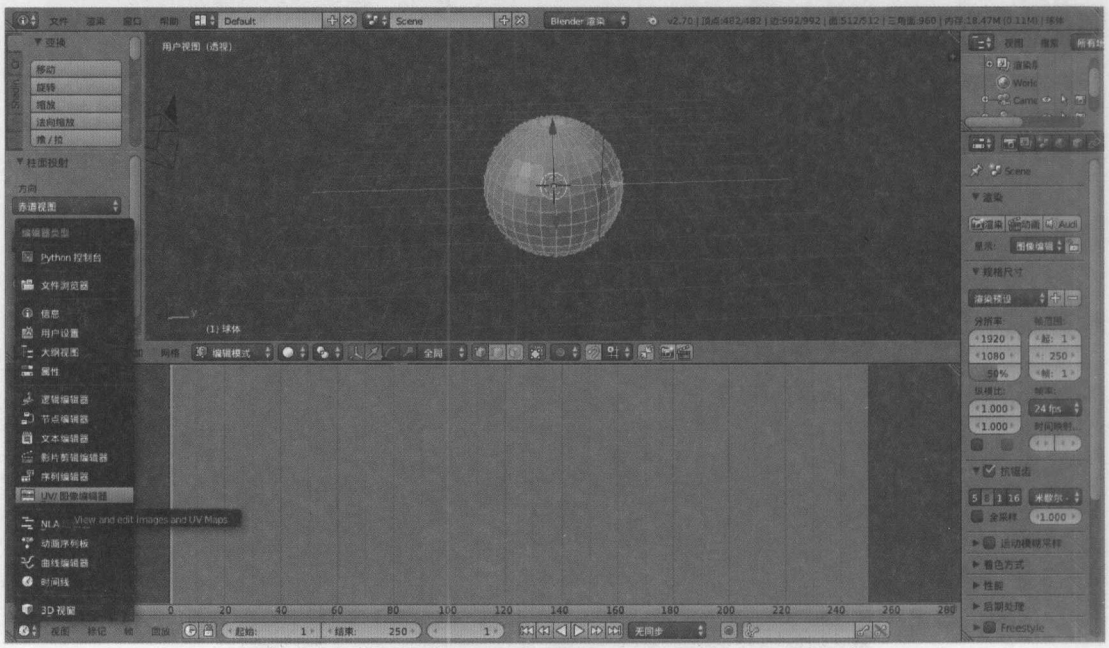


图 2-22　UV/图像编辑器

第五步：这时会看到如图 2-23 所示的 UV 展开图。

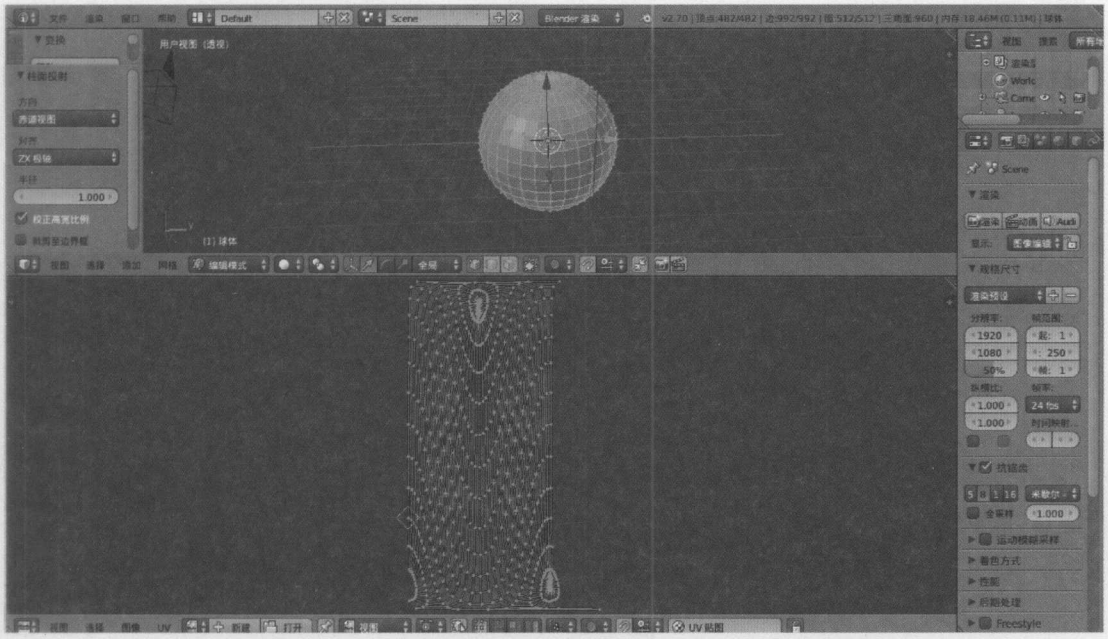


图 2-23　UV 展开图

第六步：在 UV/图像编辑器中依次单击“图像”、“打开图像”，选择事先准备好的地球图片打开，打开图片后效果如图 2-24 所示。

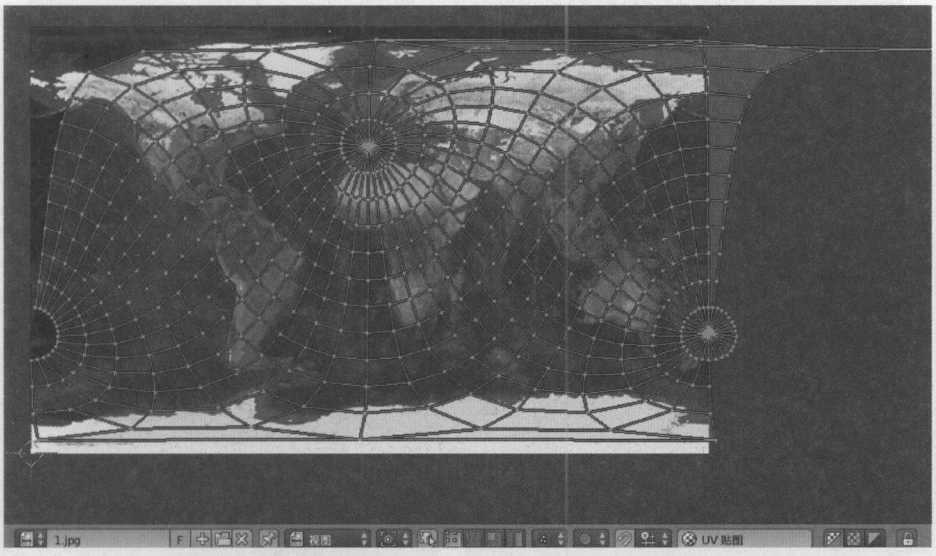


图 2-24　UV 展开图效果

第七步：调整网格，使网格正好适合图片的大小。选中整个网格，按“S”键，拖动鼠标可以调整网格的比例大小；按“R”键可以旋转整个网格；按“G”键可以移动整个网格。也可以选中一部分网格，进行调整。按“C”键可以用圆形圈选，按“B”键是矩形圈

选。调整好的网格如图 2–25 所示，网格正好覆盖住图片。

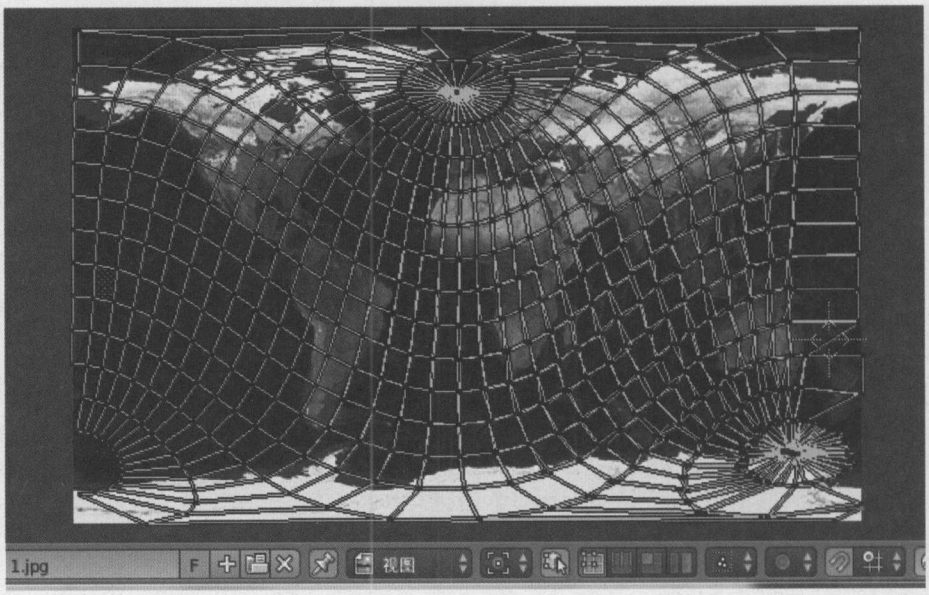


图 2–25　调整网格

第八步：在右侧的纹理面板中，纹理类型选择“图像/影片”，并选择刚才这张地球的图片，如图 2–26 所示。

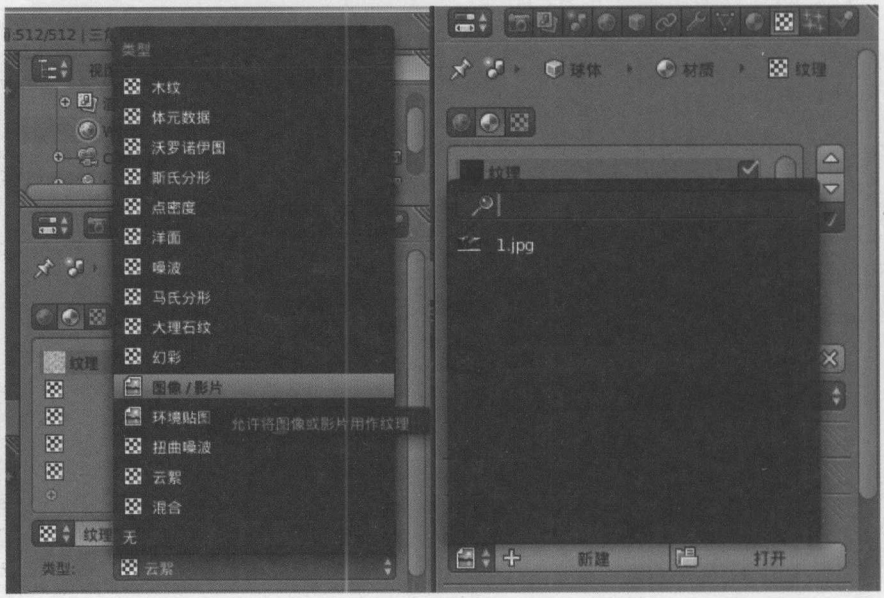


图 2–26　纹理面板

第九步：在“纹理”一栏中“投射模式”选择“UV”选项，如图 2–27 所示。

第十步：打好光照，光源的位置如图 2–28 所示。

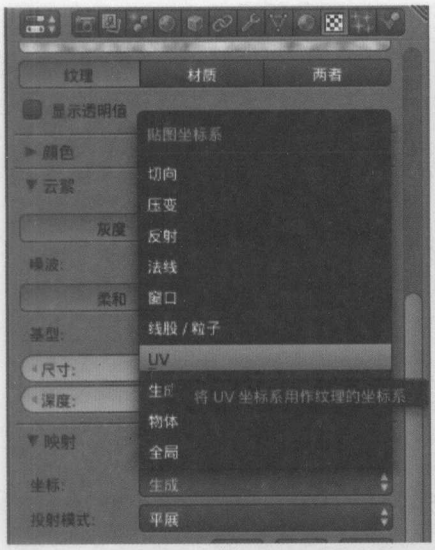


图 2-27　投射模式

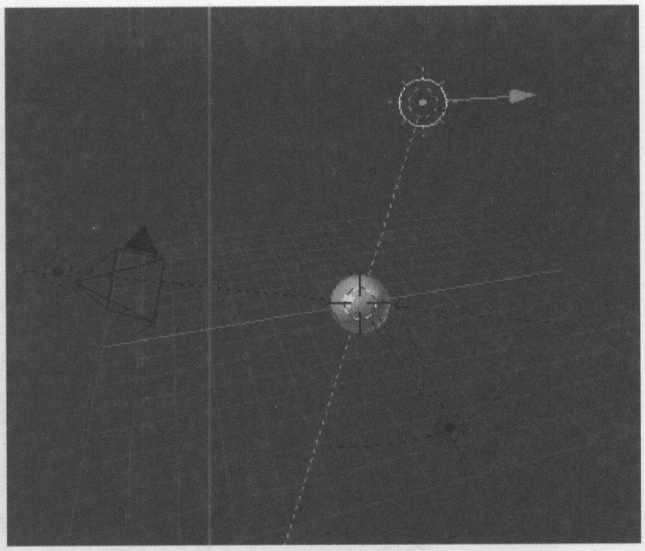


图 2-28　光照

第十一步：最后渲染出图（按“F12”键），如图 2-29 所示。

2. 另一种方法

上述的步骤采用的是 Blender 提供的自动展开方法，没有手工对封闭的球形进行切开和展开。还有一种方法是手工对球形切开，步骤如下。

第一步：删除立方体，添加一个经纬球模型，单击菜单添加—网格—经纬球。

第二步：按“Tab”键进入编辑模式，选择边编辑模式，按住“Shift”键用鼠标右键点选一圈线，如图 2-30 所示。

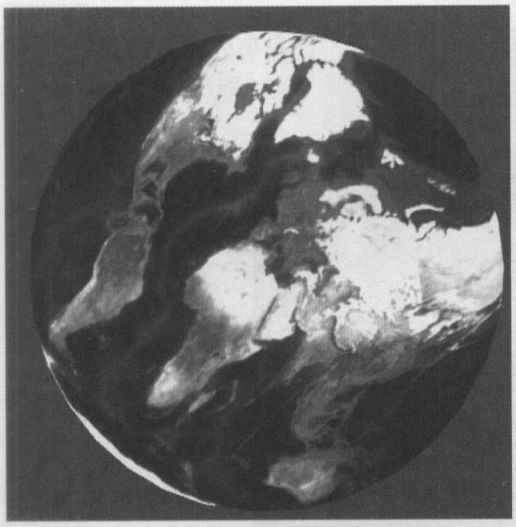


图 2-29　渲染出图

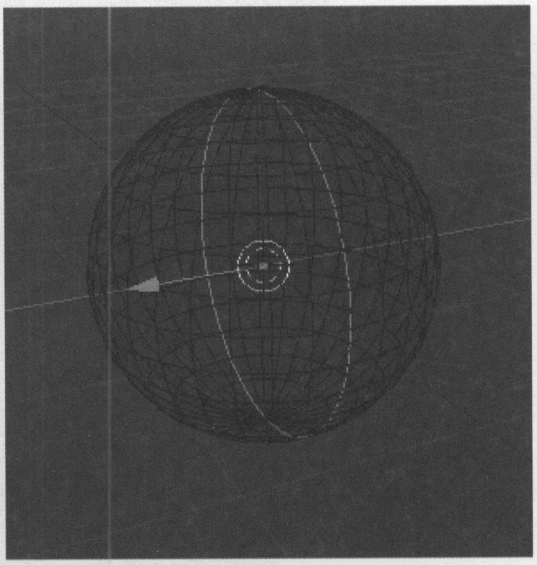


图 2-30　编辑模式

第三步：按“CTRL + E”组合键，选择标记缝合边，然后按“A”键全选网格，再按“U”键选择“展开”，如图 2-31 所示。

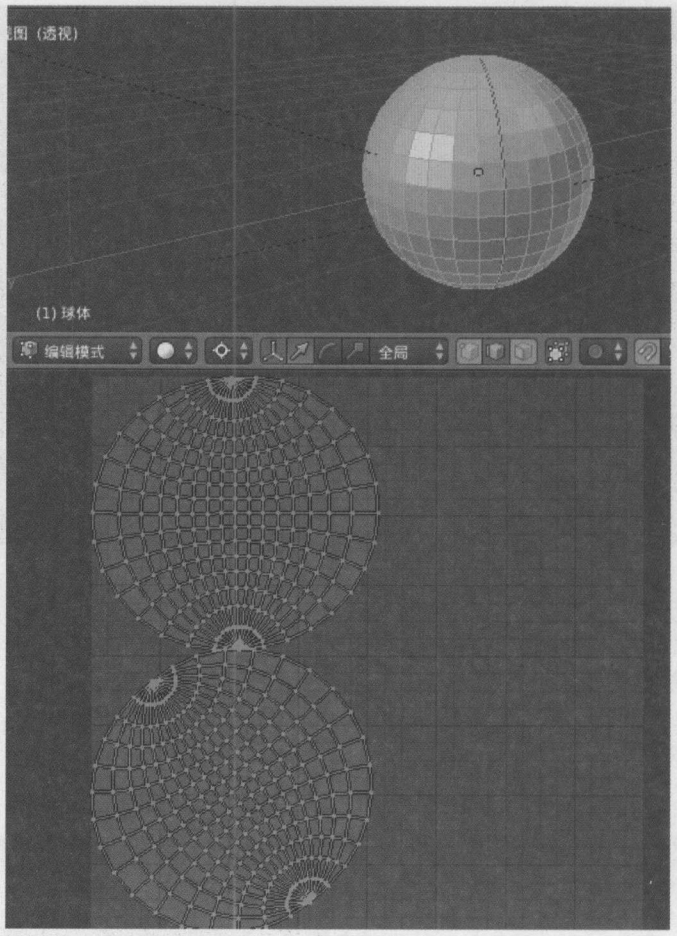


图 2-31　展开

第四步：在 UV/图像编辑器中依次单击图像 - 打开图像，选择事先准备好的图片并打开。调整网格，使网格正好适合图片上大小，如图 2-32 所示。“S”键可调大小，“G”键移动网格，“R”键可旋转网格。

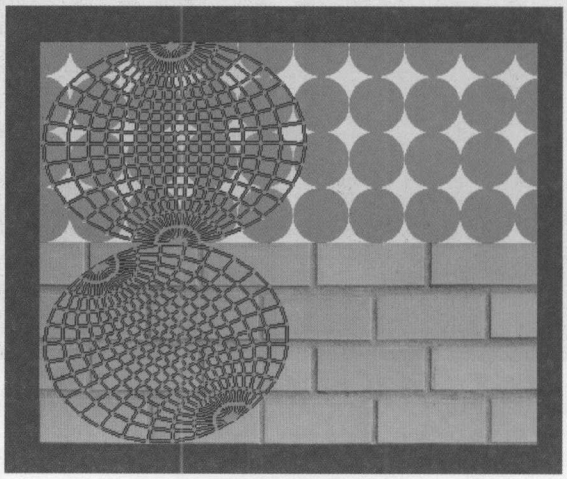


图 2-32　UV/图像编辑器

第五步：在纹理类型选择图像/影片菜单中，选择刚才的这张图片，如图 2-33 所示。

第六步：在“映射”一栏的“坐标”中选择“UV”选项，如图 2-34 所示。

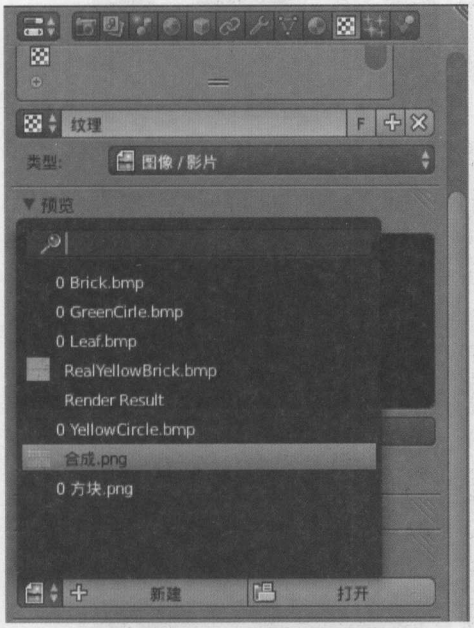


图 2-33　选择图像

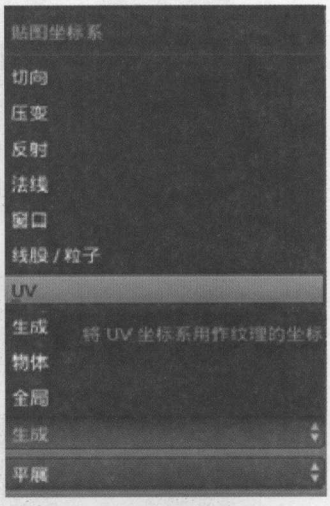


图 2-34　映射

第七步：打好光照，按“F12”键渲染出图，如图 2-35 所示。

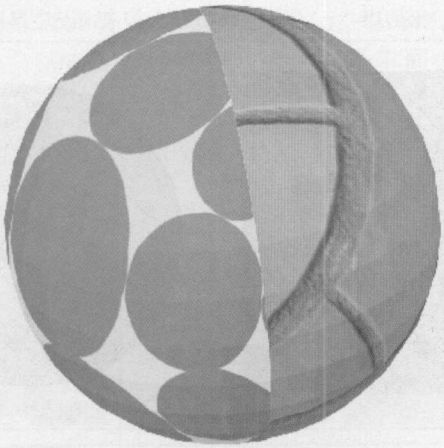


图 2-35　渲染出图

3. 效果图

如图 2-36 和图 2-37 所示是不同视角、不同图像的贴图效果。由于对网格有拉伸调整，所以最后贴图效果会有些扭曲和失真，这是正常现象。一般可以把失真的部分放在视图背面隐藏起来。

如图 2-37 所示，在第一行贴图中，由于这是将球从中间剪开贴图的，所以中间会出现不能很好对接的现象。第二行贴图中是把两张图片合成到一张图片作为贴图贴到球上的效果。

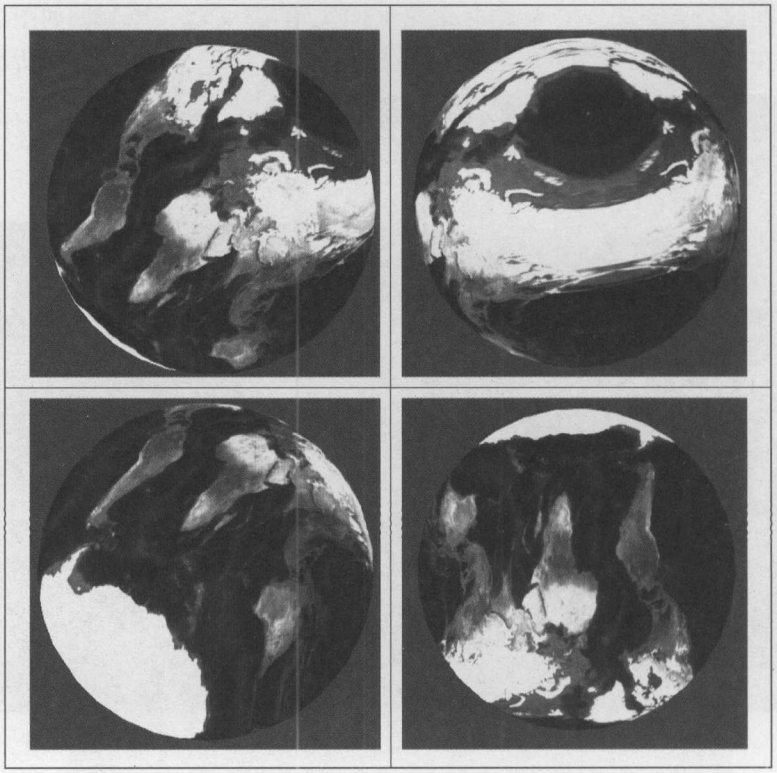


图 2-36　不同视角贴图

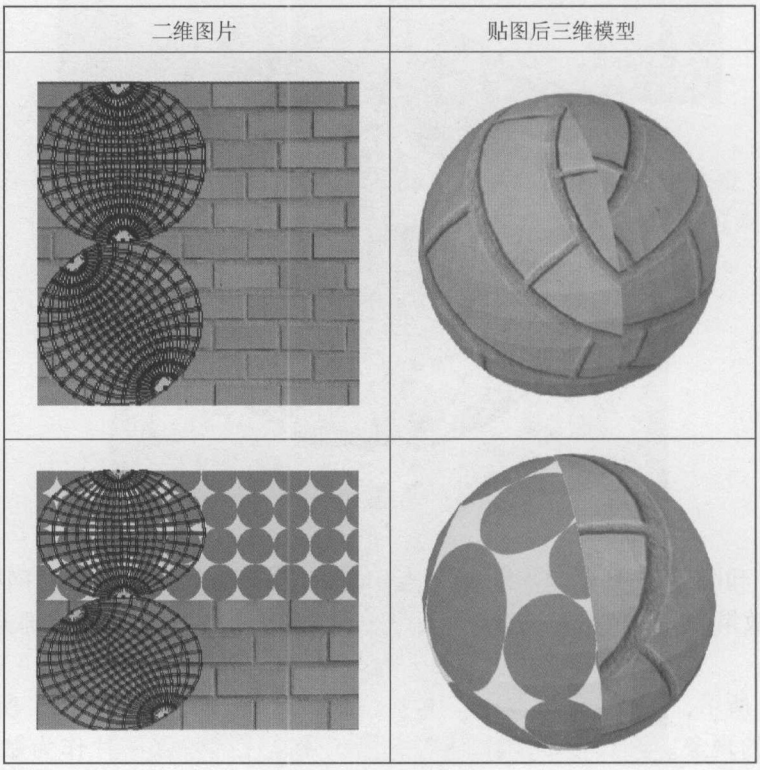


图 2-37　不同图像贴图

2.4　凹凸贴图

二维图像是个平面图像，但在渲染时，常常需要体现具有立体凹凸效果的三维模型。如图 2-38 所示一张平面经过贴图后显示凹凸不平的渲染效果。

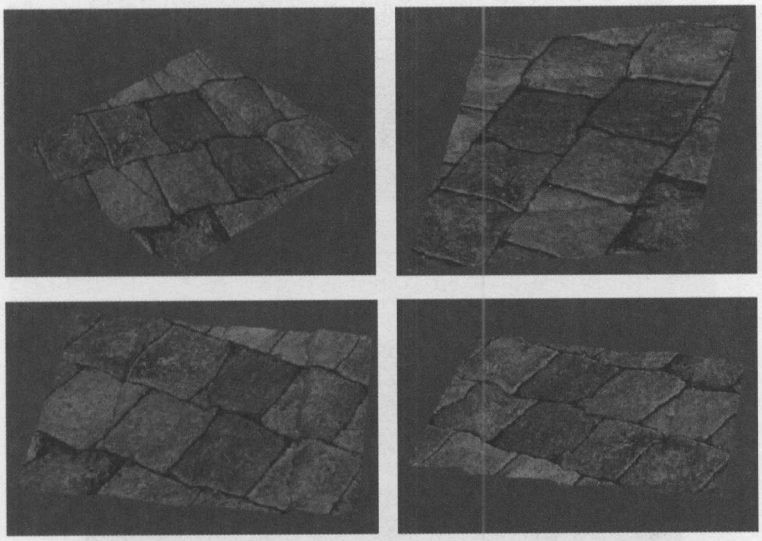


图 2-38　凹凸贴图

这种凹凸不平的渲染效果需要特殊处理的贴图，这些特殊的贴图有 Normal、Displacement、Occlusion、Specularity、Diffuse 五种类别。用这五种不同的贴图可以得到不同的最终渲染效果。使三维模型的渲染更加逼真和丰富。这五种贴图可以用普通的图片来生成，如用 Crazybump 来制作五种贴图。如图 2-39 所示是 Crazybump 软件从一张图片生成的五种不同贴图的展示。

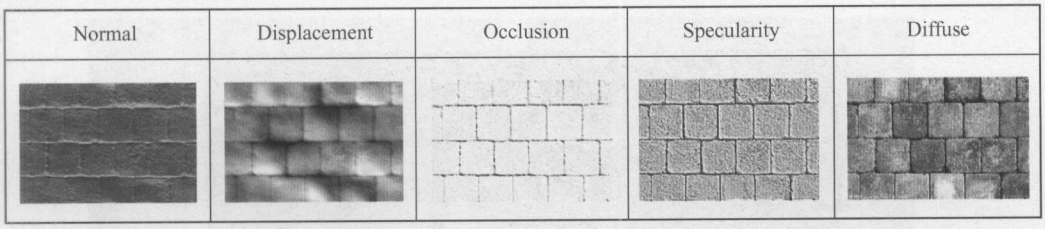


图 2-39　五种贴图

Crazybum 使用方法如下。

（1）单击“打开”按钮，如图 2-40 所示。

（2）选择一种图片格式。要根据图片实际的状态来选，这里要处理的是一张放在桌面的图片，所以选第一种，如图 2-41 所示。

（3）找到要打开的图片并选中，单击“打开”按钮。

（4）选择要产生的凹凸类型，这里选择第一种，如图 2-42 所示。

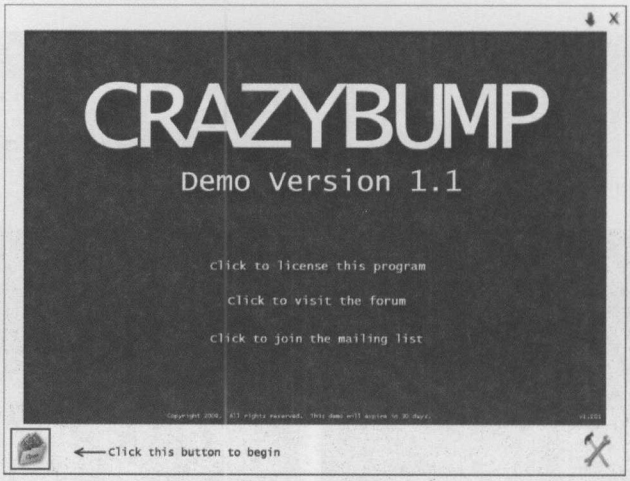


图 2-40　CrazyBump 软件

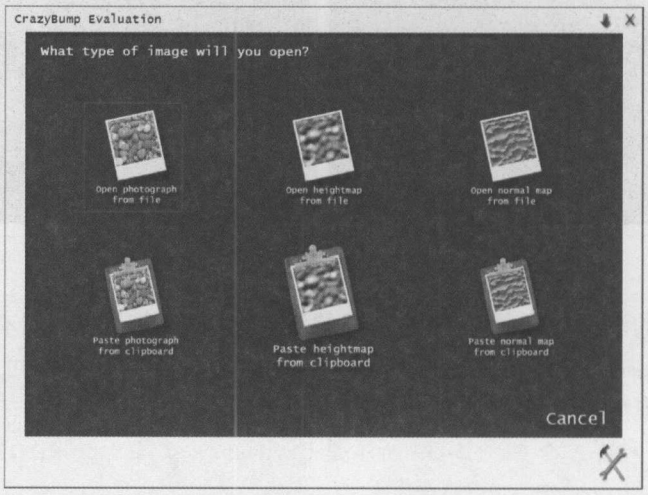


图 2-41　图片格式

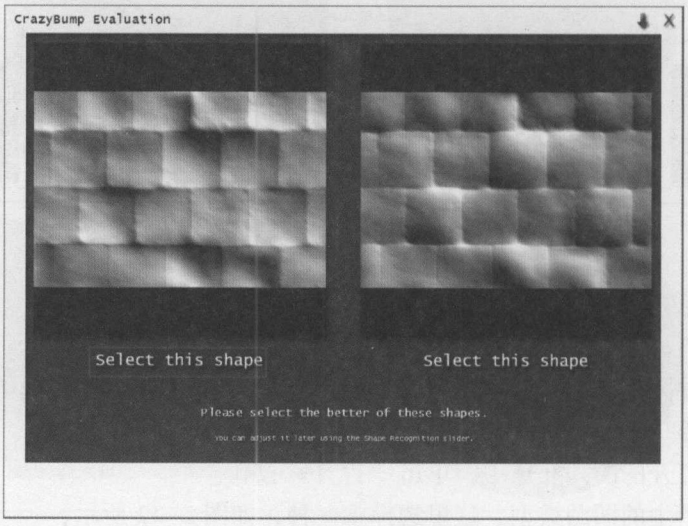


图 2-42　凹凸类型

（5）保存处理好的图片。其中有些图片处理的参数是可以根据需要来调整的，单击“保存”按钮，然后选择“保存所有纹理图”选项，如图 2-43 所示。

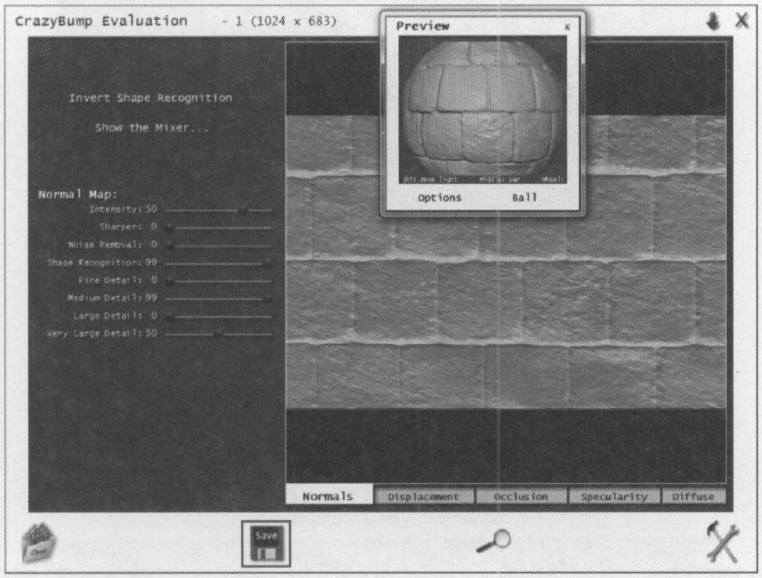
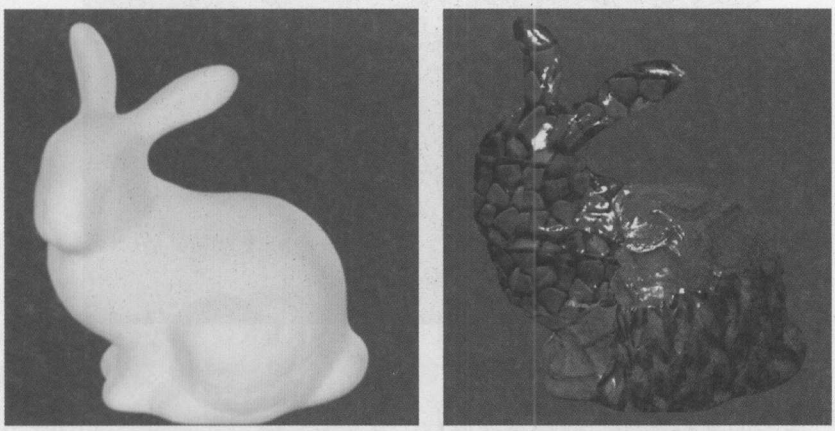


图 2-43　保存所有纹理图

2.5　兔子贴图

在贴图时肯定会遇到同一个模型上的不同部分需要贴上不同图像的情况，本节就将介绍这种贴图方法。同时会介绍如何结合用 Crazybump 软件贴出具有凹凸感效果的图。如图 2-44 所示是兔子三维模型的贴图，从中可以看出兔子的不同部位图像不一样，同时兔子的渲染效果由于使用的凹凸贴图，从而能够具有层次感和立体感觉，更加逼真。



（a）没贴图　　（b）贴图后

图 2-44　兔子三维模型的贴图

1. 兔子模型贴图步骤

第一步：删除立方体模型，添加一个兔子的模型（文件—导入—obj 文件），如图 2-45 所示。

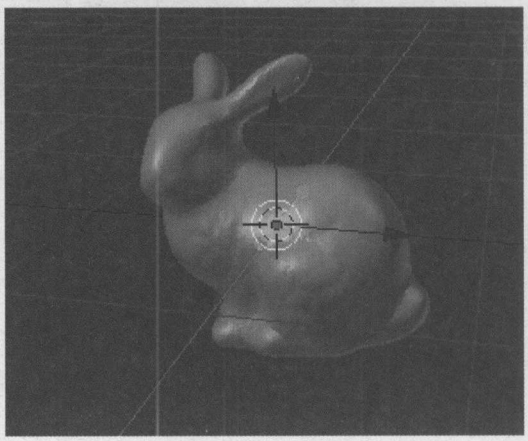


图 2-45　兔子三维模型

第二步：重复立方体例子中的第一步至第三步，得到标记好的图，将网格分成四部分，如图 2-46 所示。

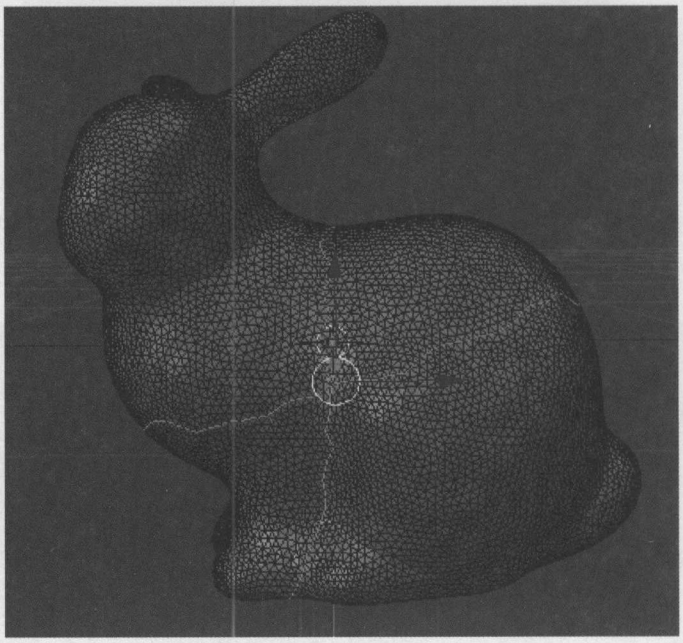


图 2-46　切开兔子

第三步：选中四部分中的一部分全部网格，按“P”键，并确定。选中时用“B”键矩形圈选，或者用“C”键圆形圈选。注意选中时一定要把每一个点都选上，否则就出现洞了，或者调成选边或选面会好选一些，如图 2-47 和图 2-48 所示。

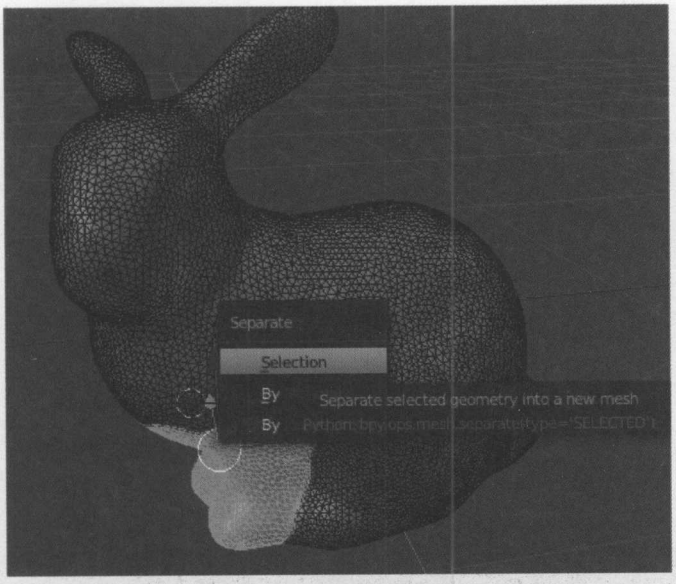


图 2-47　选择菜单

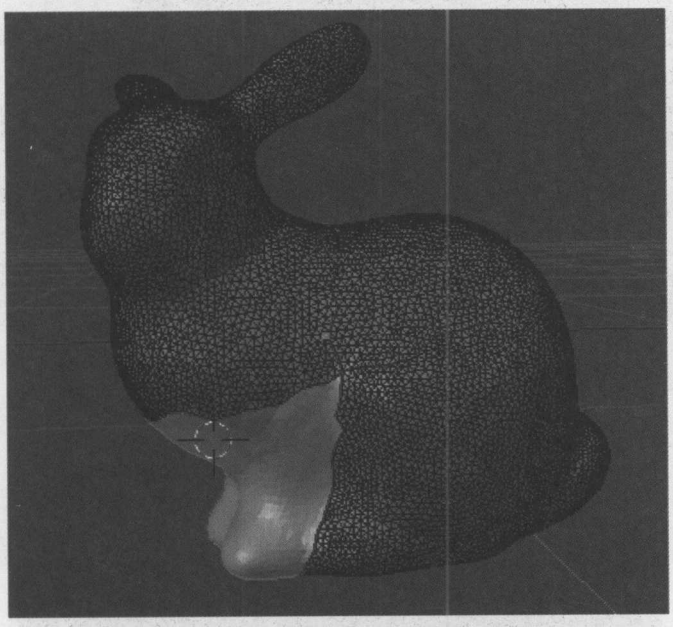


图 2-48　选择结果

第四步：重复第三步，将其他三部分也分离出来。这样模型就成功地分成四部分了，可以对每部分进行单独操作，可以贴上不同的图，如图 2-49 所示。

第五步：选取兔子头这部分进行贴图，将其他部分隐藏。隐藏的方法是在右面的列表里单击“眼睛”形状的那个图标，这个图标变暗了对应的图形就隐藏了，想显示的话再单击这个图标。“眼睛”图标旁边的“箭头”图标的作用是相应部分能否操作，单击这个图标变暗后，就不能对此部分进行任何操作了，再单击就恢复了，如图 2-50 所示。

第六步：选择 Cycles 渲染模式。Blender 提供三种渲染引擎，不同引擎的渲染效果会不同，如图 2-51 所示。

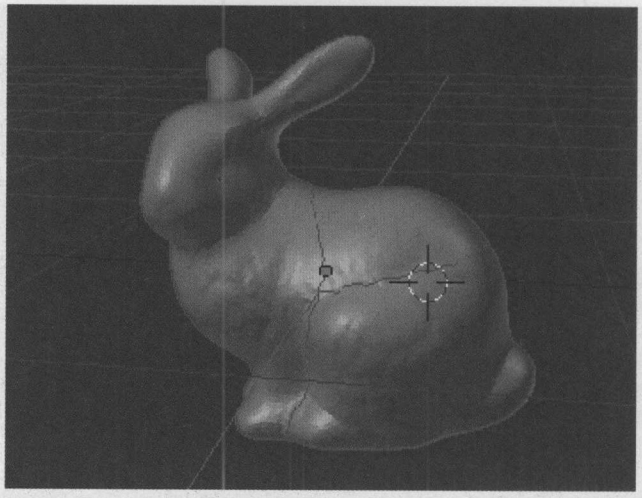


图 2-49　四部分

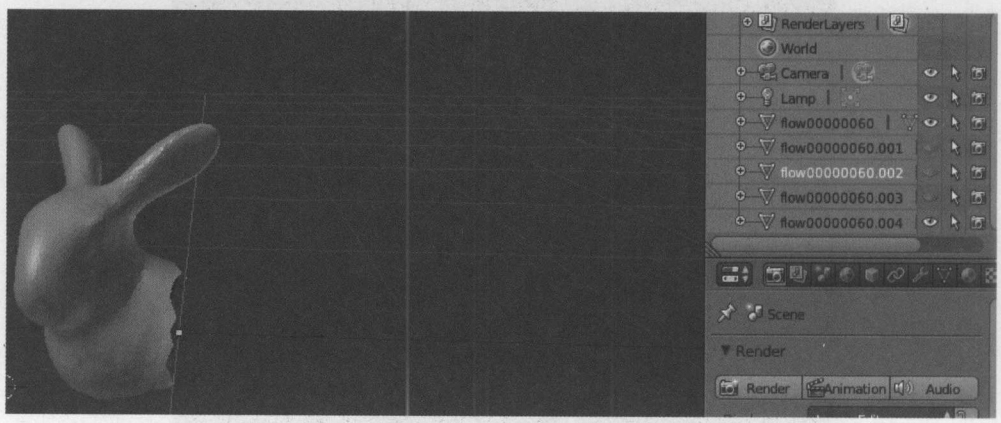


图 2-50　兔子头贴图

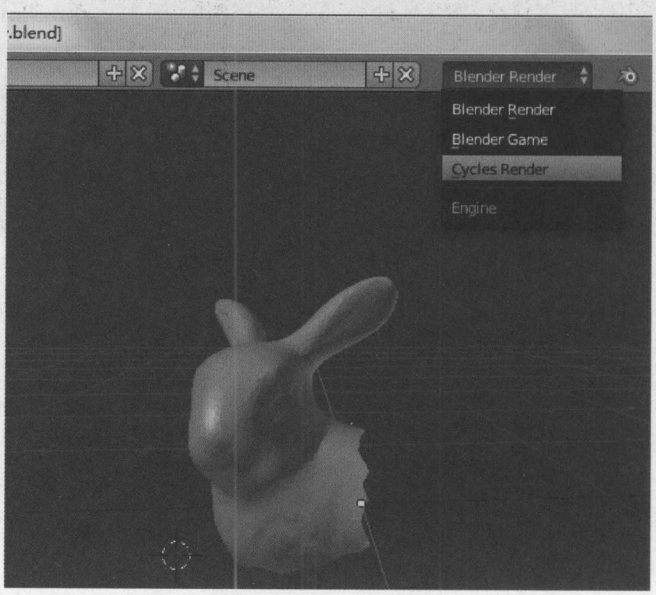


图 2-51　渲染模式

第七步：按“U”键，选择从视角投射项展开，如图 2-52 所示。

第八步：将下面的窗口改为“节点编辑器”窗口，显示该节点编辑器，如图 2-53 所示。

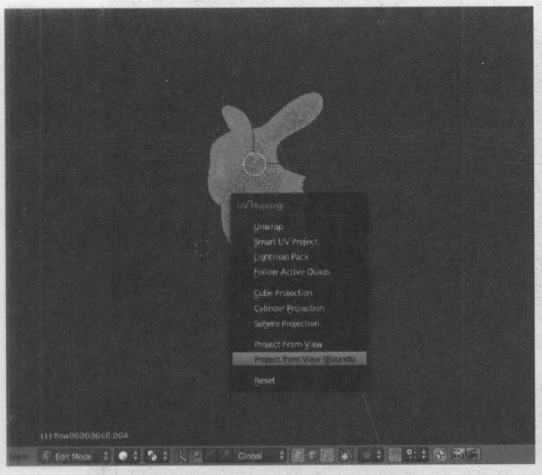


图 2-52　视角投射菜单

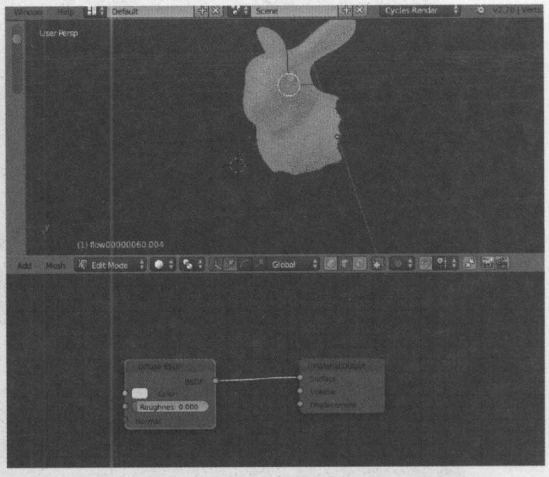


图 2-53　节点编辑器

第九步：添加图像纹理节点，单击菜单增加—纹理—纹理图像 ，并加载待贴图像，然后将其连接到漫反射颜色输入，如图 2-54 所示。

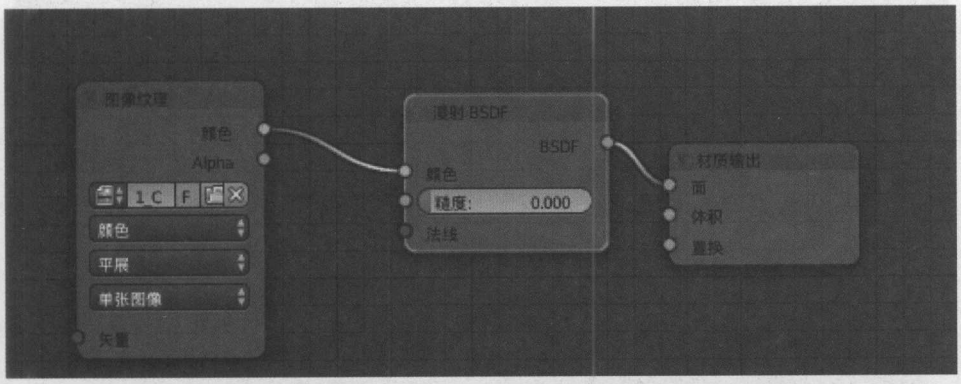


图 2-54　纹理节点

第十步：按图 2-55 所示添加各节点，并在新加的图像纹理节点打开刚才用 Crazybump 生成的 Specularity 图。

第十一步：按图 2-56 所示添加各节点，并在新加的图像纹理节点打开刚才用 Crazybump 生成的 Normal 图。

第十二步：添加一个 Displace 细分，如图 2-57 所示。

第十三步：进入到纹理面板，选择“Displace”选项，方式改为“image or movie”，然后选择前面创建的产生位移图，如图 2-58 所示。

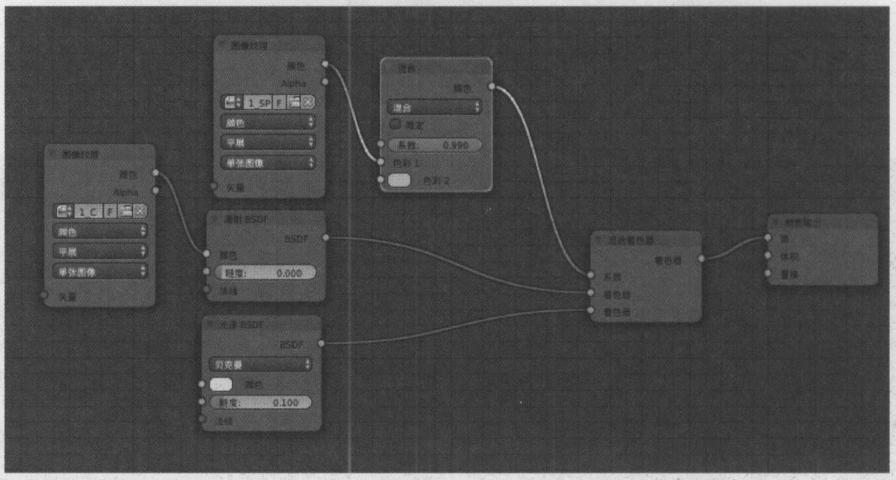


图 2-55　添加各节点

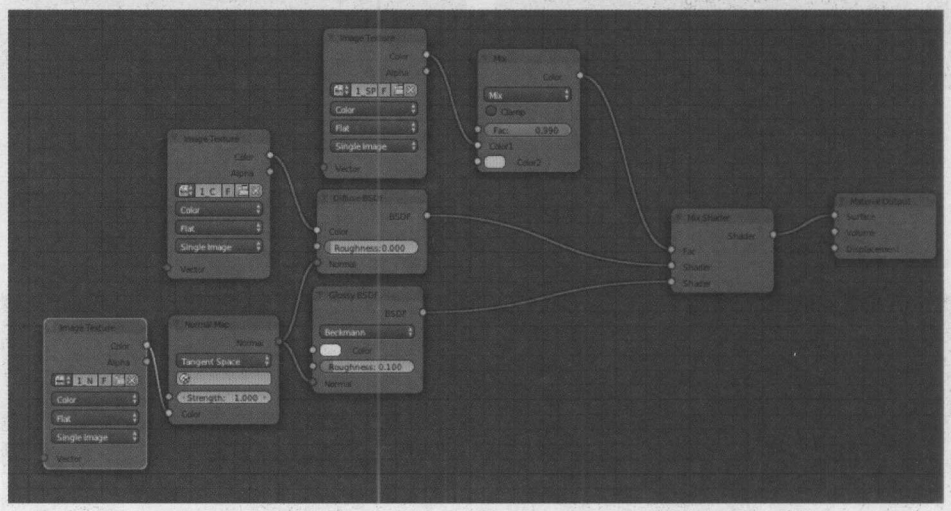


图 2-56　设置法向贴图

图 2-57　设置 Displace 细分

图 2-58　产生位移图

第十四步：返回 Modifier 面板，设置强度为“0.02”和方向为“RGB→XYZ”，如图 2-59 所示。

图 2-59　设置强度

第十五步：按图 2-60 所示添加节点，在图像纹理节点打开刚才用 Crazybump 生成的 Occlusion 图，并将加的 mix 节点方式改为 multiply。

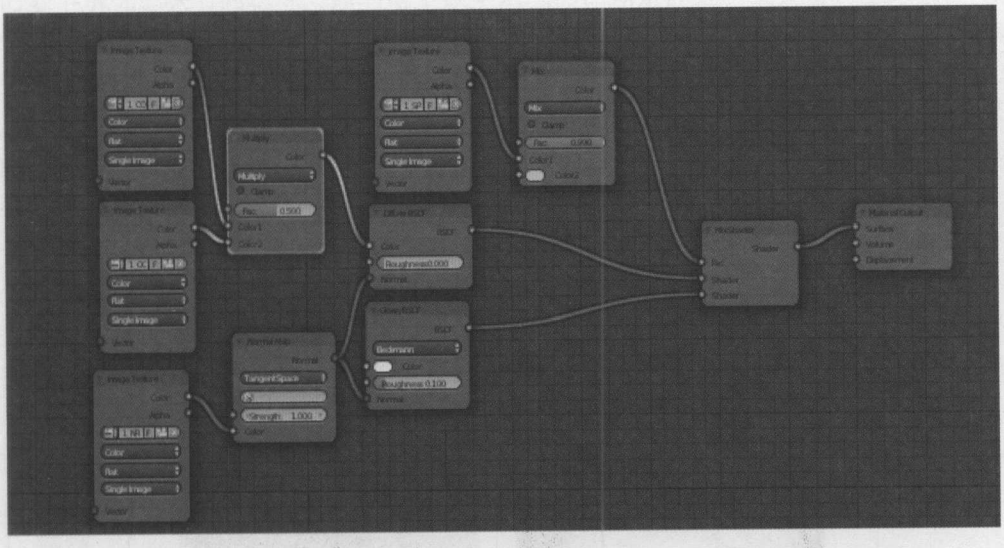


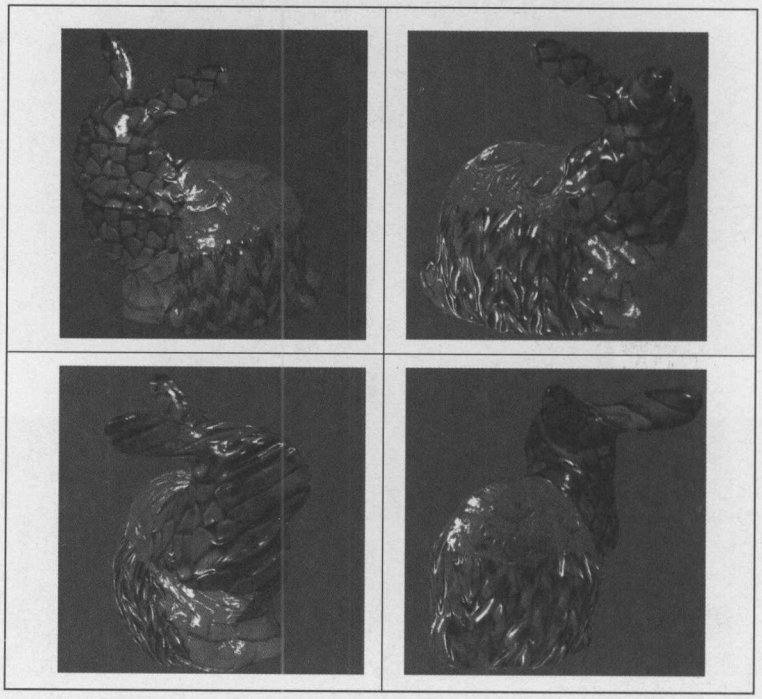
图 2-60　设置 Occlusion 图

第十六步：重复第九至第十四步，为其他三部分贴上图。也是只留要贴图的部分可见，其他部分隐藏。

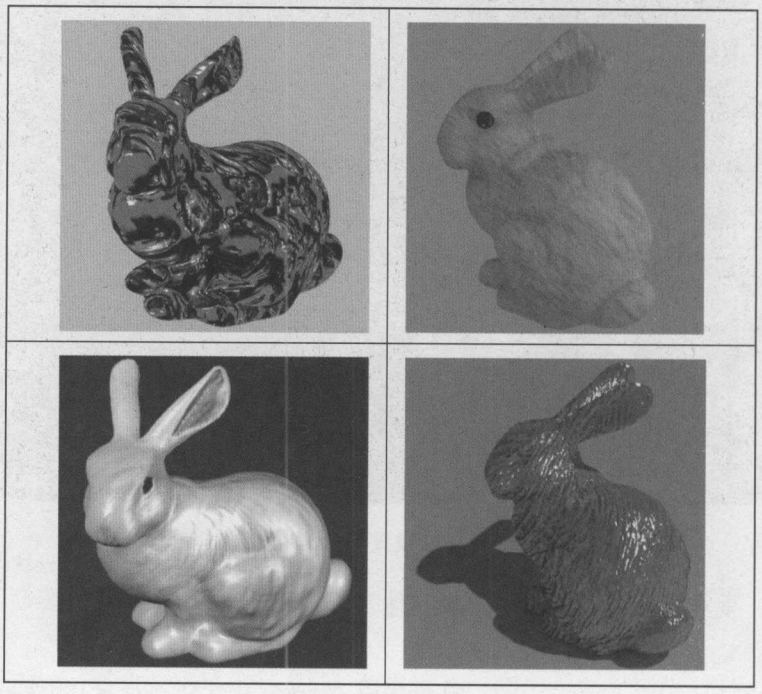
第十七步：打好光照，渲染出图。分辨率改为 4096 × 4096 像素，100%。采样中的渲染项改为 400，当然越大越好，要看计算机的承受程度来设定。

2. 效果图

如图 2-61 所示是不同视角和不同贴图图片产生的凹凸贴图效果。从中可以看出，三维兔子身上看起来具有高低起伏的不同细节，并且采用不同图片后，效果显示不一样。



（a）不同视角



（b）不同贴图图片

图 2-61　凹凸贴图效果

第3章 三维模型参数化

3.1 参数化概念

三维网格模型的参数化技术起源于把二维的纹理贴图映射到三维网格模型上的研究。现在三维网格的参数化在计算机图形学和几何模型处理等很多方面都得到了应用。三维渲染指的是为三维模型增加颜色、凹凸贴图等，从而使三维模型看起来更丰富、更真实。如果三维模型的分辨率足够大，那么可以直接在三维模型的顶点上绘制，为每个顶点绘制一个颜色。但对于大多数模型来说，是把三维模型进行参数化展开到二维平面，然后在二维平面上进行纹理贴图。把三维模型展开为二维平面模型的过程称为三维模型的参数化（Parameterization）。

三维模型的参数化是三维模型处理中一个非常重要的领域，需要大量的数学知识，如线性代数、微分几何、微积分、离散外积分等。参数化把位于三维空间的三维模型，通过构建相应的数学系统，映射到二维平面。这个数学系统的已知条件是三维模型在三维空间里的各种几何和拓扑属性，以及三维模型展开后拓扑不变，并且所有顶点展开后位于一个平面上。根据这些已知条件，通过设定的不同的限制条件，可以构建各种不同的线性或非线性的数学系统来求解每个顶点在二维平面的位置。

在本书中，三维模型用 M 表示，（V，E，F）三个集合分别表示三维模型顶点、边和面的集合。每个顶点用 v_i 表示，每个边用 e_i 表示，每个面用 f_i 表示。三维模型每个顶点有（x，y，z）三个坐标，参数化后的二维平面模型每个顶点坐标是（u，v）。三维模型参数化后每个顶点保持原来的连接关系不变，改变的是每个顶点的几何位置，从而使所有顶点都位于一个平面上。

如图 3-1 所示为几个三维模型参数化后的二维模型，从中可以看出，三维模型在参数化后仍能够保持原来模型的特征。例如，耳朵部分，在参数化后，耳朵的轮廓在二维上依然显现。而兔子头的部分，参数化后，三维模型上的顶点分布在二维上仍具有相应类似的分布。

1. 属性

参数化就是把一个曲面映射到另一个曲面，不是每个曲面都能够一对一地映射到另一个曲面。只有两个拓扑相同的三维曲面理论上可以进行一对一的映射。如果其中三维曲面用网格来表示，那么这个映射称为网格的参数化（Mesh Parameterization）。其中三维网格映射到的曲面称为参数化域（Parameter Domain）。根据参数化域的不同和预期的参数化效果的属性不同，有很多不同的进行参数化的算法。不同的参数化算法的质量（Quality）、效率（Efficiency）、稳定性（Robustness）不同。

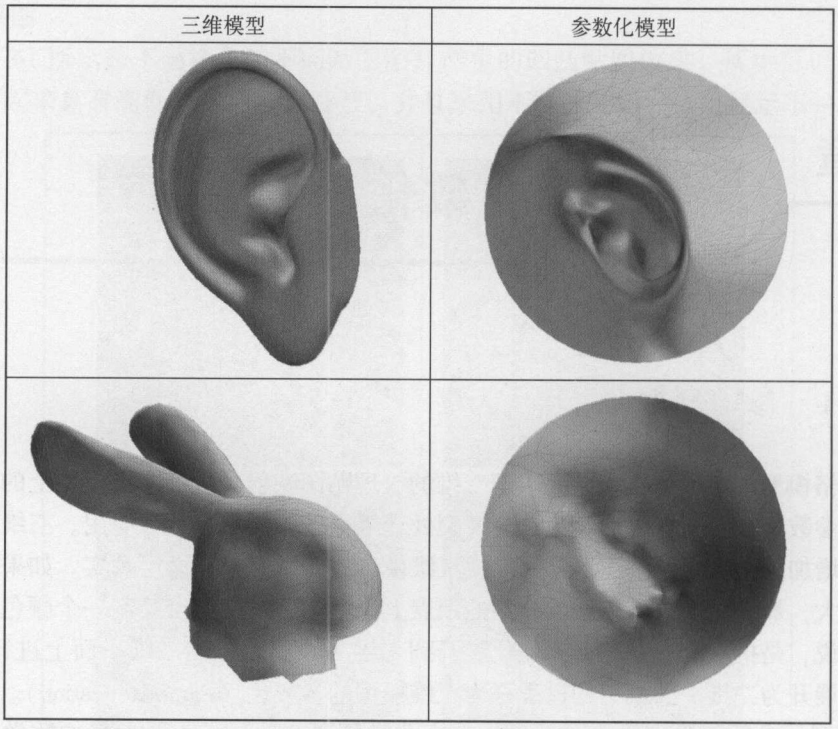


图 3-1　三维模型参数化

三角形网格模型由若干个三角形面组成。三角形网格参数化的目的是获得此三角形网格在一个特定的参数化域的三角形化（Triangulation）的映射。也就是把网格模型上的每个三角形映射到参数化域上的三角形。这个映射是分段线性的（Piecewise Linear）。网格上的每个顶点对应到平面上一个二维点。

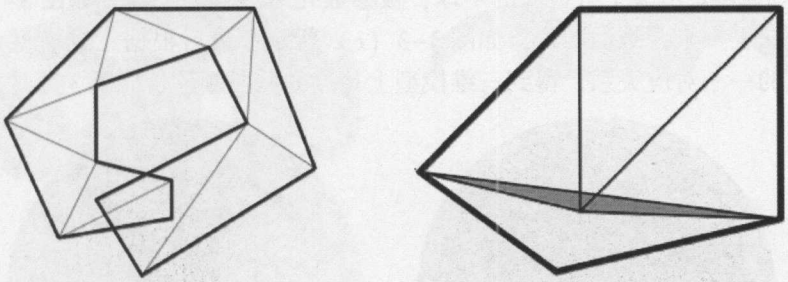
2. 双射

参数化的一个重要目标是获得一个双射（Bijective）的映射。也就是映射后，参数化域上的每个点对应且只对应网格上的一个点。双射可以分为局部双射（Local Bijective）和全局双射（Global Bijective）。局部双射只需要满足网格的一个小区域内映射是双射的条件，也即是一个三角形和其邻居的三角形范围内是双射的，这样在全局上允许出现重叠（Overlap）。如图 3-2（a）所示不是一个双射，而是一个局部双射，而图 3-2（b）所示不是一个局部双射，其中一个三角形和周边的三角形法向不一样，出现了翻转。

3. 扭曲

映射后的域三角形（Domain Triangle）和源三角形（Original Triangle）在形状上通常会发生变化，从而产生角度扭曲（Angle Distortion）和面积扭曲（Area Distortion）。参数化映射的目标是使整个网格上的所有三角形这些扭曲之和最小化。对于平面映射来说，大多数网格模型的参数化都会造成扭曲，只有可展面（Develop Surface）的网格模型才能获得没有扭曲的参数化映射，如圆柱（Cylinder）、圆锥（Conical Sheet）等。这种没有扭曲的参数化称为是距参数化（Isometric Parameterization）。使角度扭曲最小化的映射称为保角参数化（Conformal Parameterization），使面积扭曲最小化的映射称为保积参数化（Authalic Parameterization）。保角参数化与和谐（Harmonic）参数化有类似的地方，但两者并不完全一样。等距

参数化（Isometric Parameterization）是保距（Distance Preservation）。



（a）局部双射　　（b）翻转

图3-2　双射示例

根据**黎曼定理**，任何两个拓扑相同的光滑曲面之间都能够满足保角映射。但黎曼定理在离散网格模型上不成立。因为对于三维网格曲面来说，每个顶点相邻的三角形角度之和小于2π，而在平面上，此角度之和一定等于2π。

单独使用保面积的参数化可能形成极端的角度和线性扭曲，因此通常都是和保角参数化一起使用的。除了保角和保积以外，其他的参数化结果扭曲的度量还有距离。每个三角形的线性映射可以分解为三部分：位移、旋转、沿着两个正交轴的不一致的缩放。根据网格模型被参数化后的应用的不同，可以选择对不同的度量进行最小化。

各种参数化算法的目的和设计方法就是对这些扭曲度量进行最小化，从而得到参数化映射的结果。根据参数化算法的复杂度，还可以把这些算法分为线性（Linear）、非线性（Non－Linear）和两者混合（Hybrid）的算法。非线性的方法虽然有理论上的支持来保证参数化结果的质量，但计算复杂度很高。混合方法就是把这些非线性方法进行线性简化，从而提高了速度。

3.2　纹理题图

纹理贴图是三维模型参数化的一种最重要的应用。三维模型在构造出来后是没有贴图的，为了表示三维模型的细节，可以通过两种方法：一种是把所有细节都建立到模型上，这样的话需要三维模型密度很大，顶点数很多；另一种方法是建立一个简单的模型，然后用一个图片来表示模型的细节。把三维模型贴上纹理是三维模型建立的一个重要的环节，所有的三维建模软件都具有这一功能，基本上采用的是相同的参数化算法。纹理贴图过程就是把图片对应到二维平面上，然后三维网格上顶点的颜色就是参数化后二维平面上对应的图片点的颜色。

用参数化方法进行贴图的过程步骤如下。

（1）把三角形网格参数化到平面，可以是圆形或正方形区域。

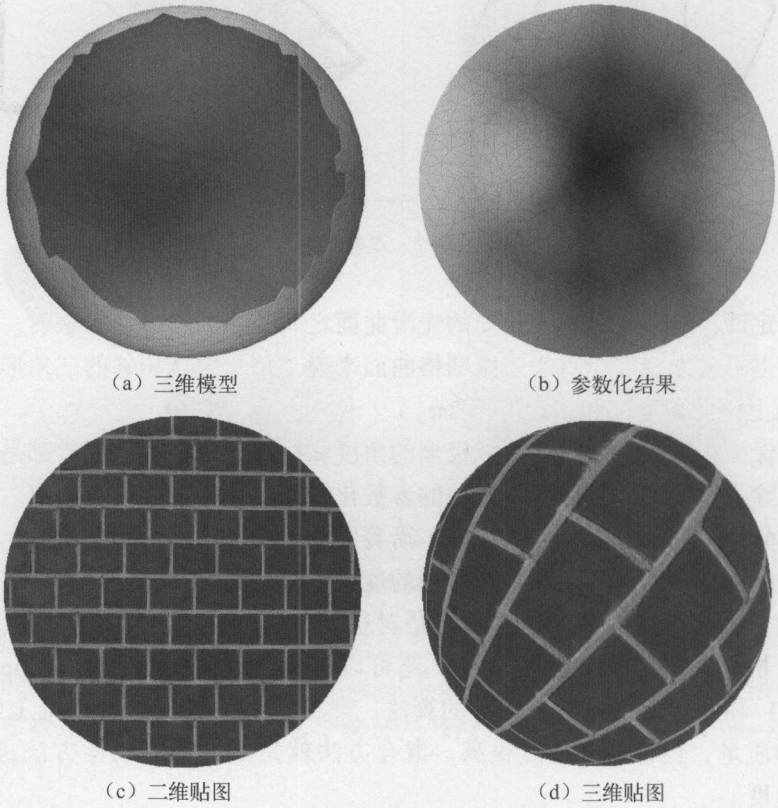
（2）把贴图和平面区域对应。

（3）三维网格上顶点的颜色对应在平面上映射点的颜色。

（4）在三维模型渲染时显示相应图片上点的颜色。

1. 贴图实例一

如图 3-3（a）表示一个有边界的半球；被参数化后展开为平面，如图 3-3（b）所示；然后这个平面贴上一个二维的图片，如图 3-3（c）所示；最后根据三维模型上的顶点和二维平面上顶点的一一对应关系，得到三维模型上顶点的贴图颜色，如图 3-3（d）所示。



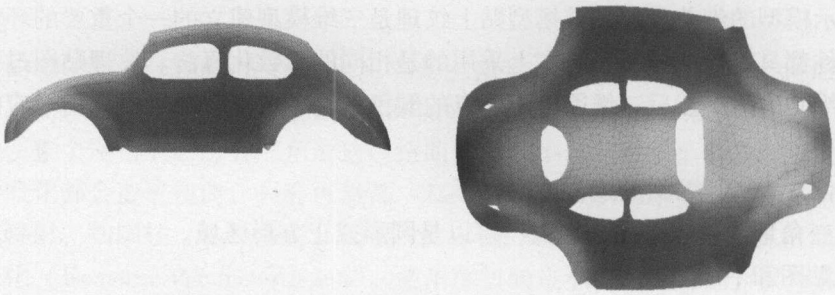
（a）三维模型　（b）参数化结果

（c）二维贴图　（d）三维贴图

图 3-3　半球三维模型纹理贴图

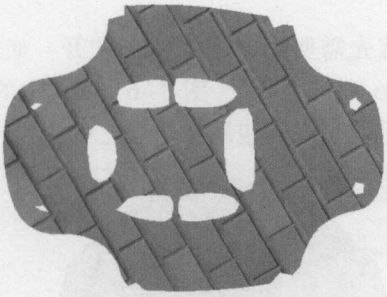
2. 贴图实例二

如图 3-4 所示是有多个边界的三维模型的贴图过程。从中可以看出，参数化把多个边界的三维模型展开后，还能够很完善地保持原来边界的形状。但参数化算法有很多种，不是每一种都能够处理多个边界的三维模型。三维参数化要求输入的模型有边界，可以是一个边界或者多个边界。假如原来的三维模型是封闭的，就需要先把原来的三维模型切开，再进行参数化。



（a）三维模型　（b）参数化结果

图 3-4　汽车三维模型纹理贴图



（c）二维贴图

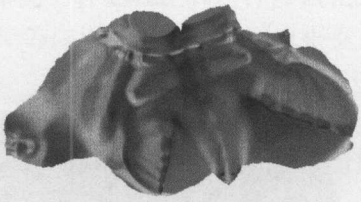
（d）三维贴图

图 3-4　汽车三维模型纹理贴图（续）

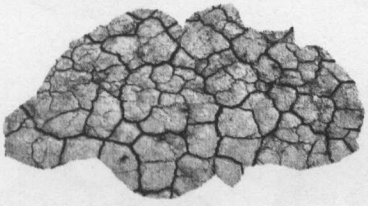
3. 贴图实例三

如图 3-5 所示是比较复杂的三维模型的贴图效果，从中可以看出，三维模型在展开后，虽然整体形状变成平面，但局部细节还能够保持，也就是原来怪兽鼻子、耳朵等部位的细节在平面上还能够具有相似的分布。

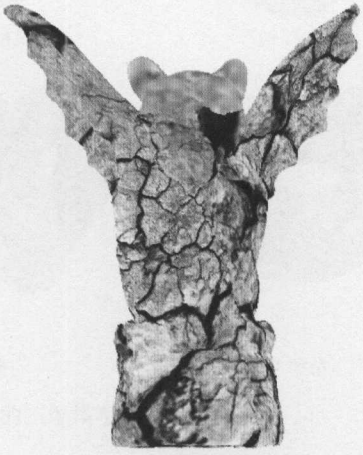
（a）三维模型



（b）参数化结果



（c）二维贴图

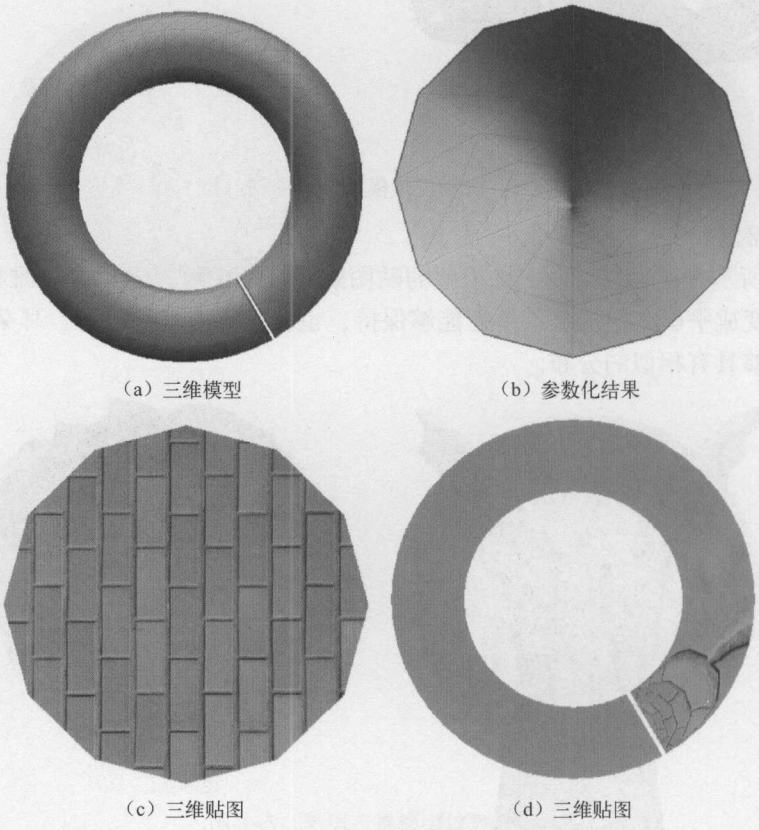


（d）三维贴图

图 3-5　怪兽三维模型纹理贴图

4. 贴图实例四

如图 3-6 所示是封闭圆环的参数化贴图。首先需要把封闭的圆环切开，如图中绿色线条所示的切开部位。然后展开为平面，如图 3-6（b）所示，图 3-6（b）中的绿色边界、绿色线条就对应着图 3-6（a）中的绿色线条。其他顶点被包围在边界之中。



（a）三维模型　（b）参数化结果

（c）三维贴图　（d）三维贴图

图 3-6　圆环三维模型纹理贴图

第4章 碟形网格参数化算法

4.1 模型拓扑

参数化算法不是对于所有模型都能够使用，一般是应用在碟形拓扑网格模型上，也就是这一类三维网格模型才能够被参数化算法从三维模型展开到二维平面。把碟形拓扑（Topological Discs）网格模型映射到一个平面上是最简单的一种参数化。这种参数化开始应用于纹理映射，现在也应用于法向、光照参数、模型压缩等应用。三维模型可以根据拓扑和边界进行分类。

拓扑结构是三维模型的一种性质，每个模型都有一定的拓扑结构，拓扑可以用边界和亏格来表示。直观地看，三维模型的亏格就是三维模型上的洞的数量。例如，封闭的球亏格是0，圆环亏格是1。不同拓扑的三维模型如图4-1所示。

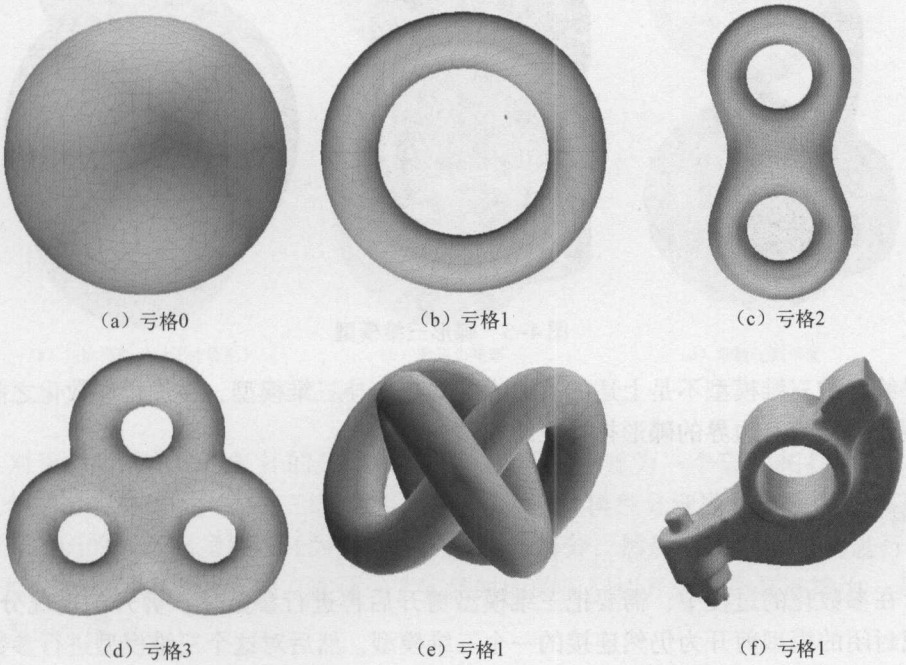
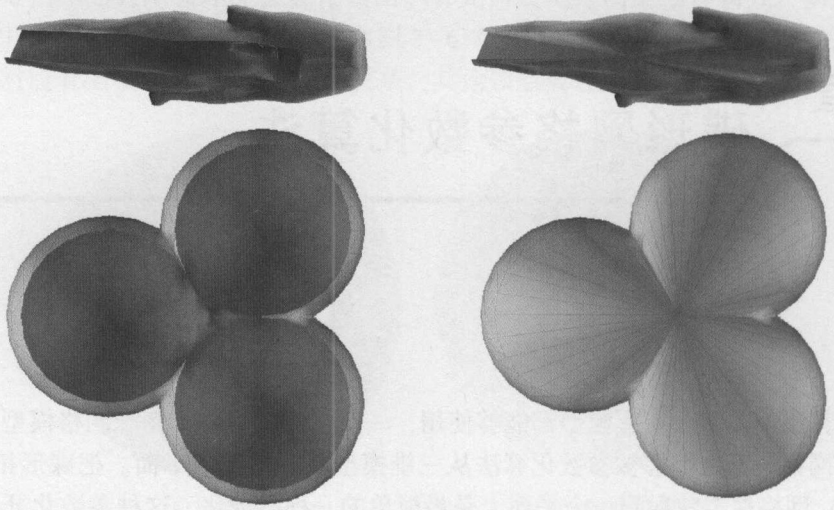


图4-1 不同拓扑的三维模型

有边界的三维模型指的是不封闭的三维模型，也就是有一些三维模型的边只有一个面。如图4-2所示是一些亏格为1，有边界和无边界的三维模型对比实例。其中蓝色的是三维模

型的外表面，橘红色的是三维模型的内表面。



（a）有边界　　（b）无边界

图 4-2　有边界和无边界的三维模型

但要把三维模型映射到二维平面，需要三维模型的拓扑和二维平面的拓扑一致，也就是需要三维模型是一个有边界的碟形拓扑，如图 4-3 所示。

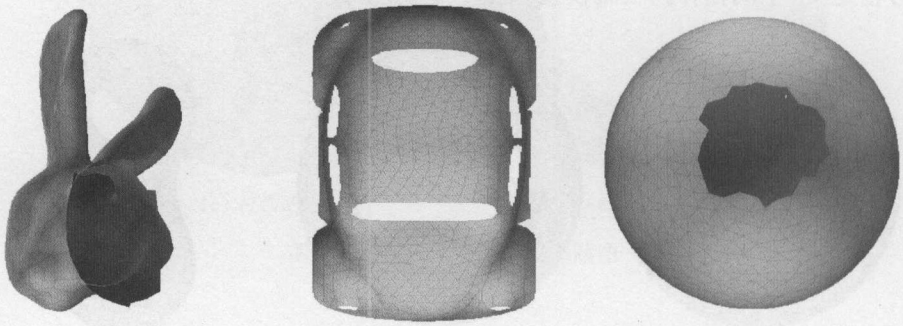


图 4-3　碟形三维模型

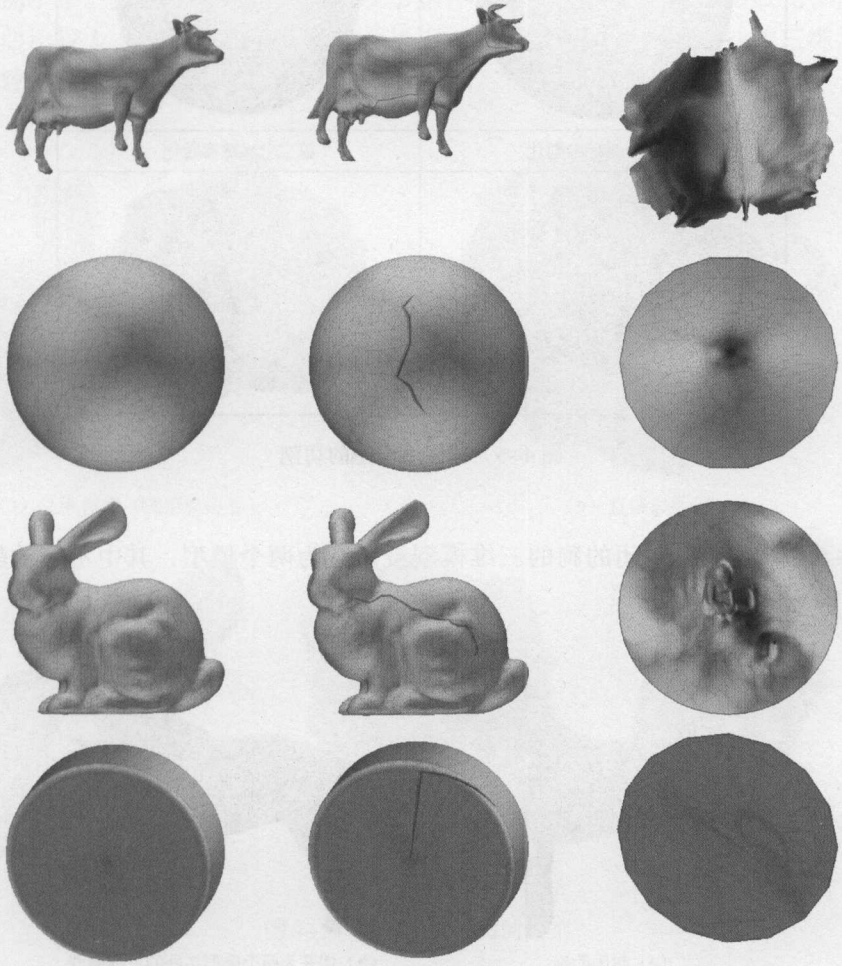
如果给定的三维模型不是上述的有边界的碟形拓扑三维模型，在进行参数化之前需要把这些模型剪开变为有边界的碟形拓扑三维模型。

4.2　模型剪开

通常在参数化的过程中，需要把三维模型剪开后再进行参数化，剪开的情况分为两种：一种是把封闭的模型剪开为仍然连接的一个三维模型，然后对这个三维模型进行参数化；另一种是把模型剪开为互不连接的几个三维模型，然后分别对这几个三维模型分别进行参数化。

（1）对于不是碟形的网格模型通常需要先把模型剪开为碟形拓扑再进行参数化。如

图 4-4 所示，原始的三维模型是封闭没有边界的，如图 4-4（a）所示，需要把这个模型表面画线，然后把模型切开，从而得到一个有边界的碟形模型，如图 4-4（b）所示，然后再参数化展开到平面，如图 4-4（c）所示。其中绿色的线条就是剪开的线条，这些线条构成剪开后的三维模型的边界。



（a）三维模型（封闭非碟形）　（b）剪开为碟形　（c）参数化到平面

图 4-4　剪开三维模型

（2）对于封闭的非碟形拓扑的三维模型，除了可以剪开为一个碟形拓扑之外，还可以直接把这个三维模型剪开为几个三维模型，并对每个三维模型分别进行参数化。如下面实例所示。这些封闭的三维模型被切开为两个互不连接的部分，然后对两部分分别进行参数化，展开为平面。和上一个实例不同，这是一个实例封闭的模型切开后，还是连接为一部分，而不是分开为两个模型。

实例一

一个封闭的球切开为两个半球，如图 4-5 所示，绿色的线条为一个封闭的线条，沿着这个封闭的线条进行切割，就把球切开为两个互不相连的半球。然后两个半球分别进行参数化展开为平面。

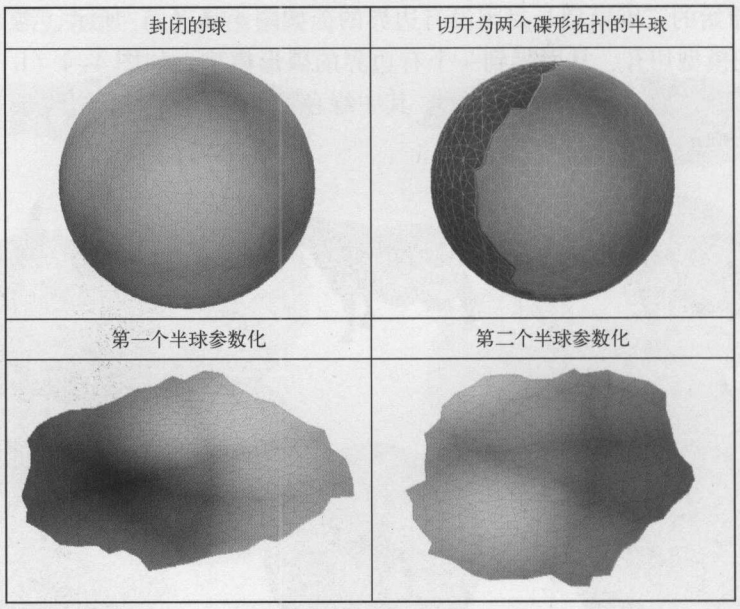
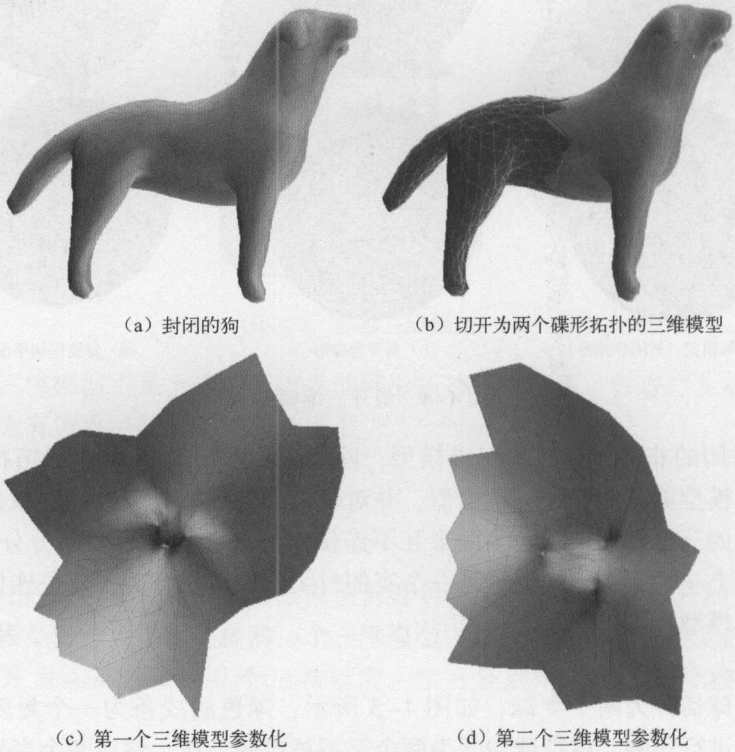


图 4-5　球三维模型的切割

实例二

如图 4-6 所示是一个封闭的狗的三维模型被切开为两个模型，其中不同的颜色表示狗的互不连接的不同的部位。



（a）封闭的狗　（b）切开为两个碟形拓扑的三维模型

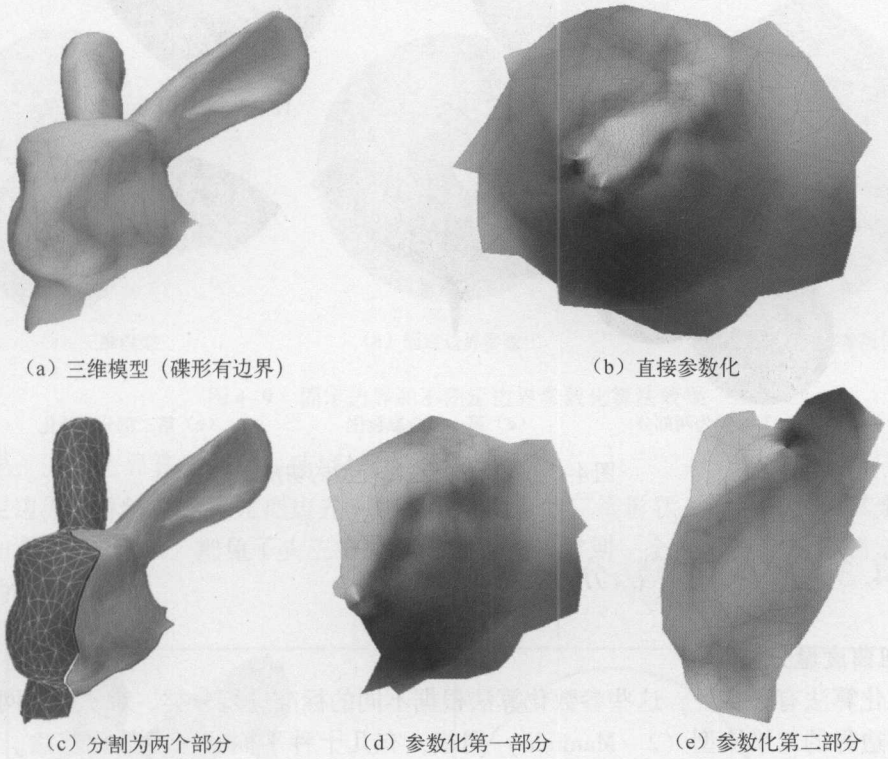
（c）第一个三维模型参数化　（d）第二个三维模型参数化

图 4-6　狗三维模型的切割

（3）即使一个三维模型拓扑结构是碟形的三维模型，但假如三维模型和平面差别太大，扭曲很大，直接进行参数化后效果不好，因此需要把这个三维模型再进行切割为几部分近似于平面的三维模型，再分别对着几部分进行参数化展开。

实例三

在这个例子中，兔子模型是有边界的碟形拓扑模型，可以直接进行参数化，但由于兔子耳朵部分扭曲比较大，参数化成平面后，效果不好。分割为两个碟形拓扑的三维模型后，再分别进行参数化后的扭曲就比较小，如图 4-7 所示。



（a）三维模型（碟形有边界）　（b）直接参数化

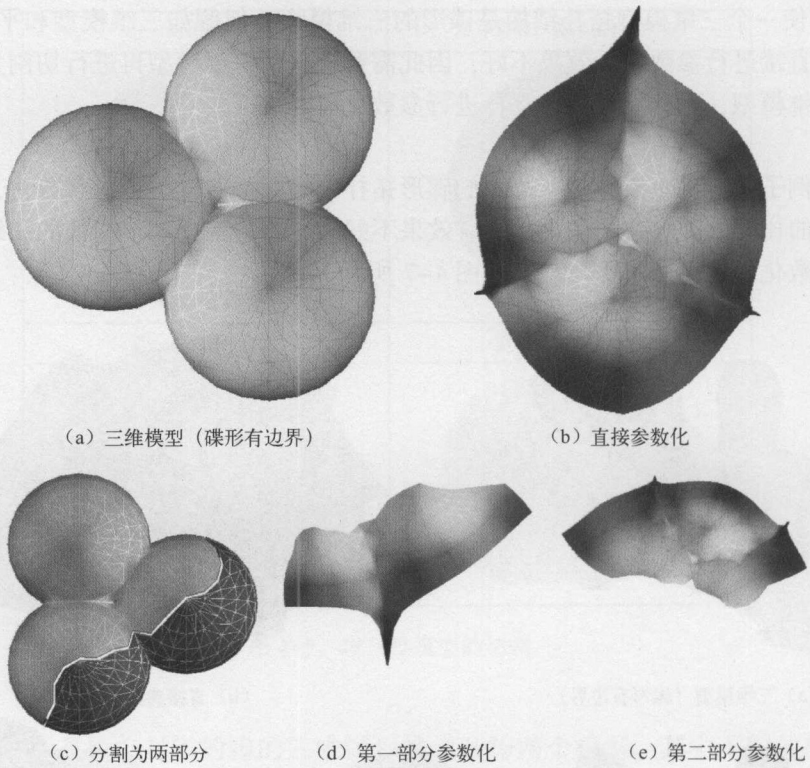
（c）分割为两个部分　（d）参数化第一部分　（e）参数化第二部分

图 4-7　兔子三维模型的切割

实例四

在这个例子中，把一个具有三个半球相连的有边界碟形拓扑的三维模型分别进行直接参数化和剪开后再参数化。得到不同的参数化结果，如图 4-8 所示。从中可以看出，切开后再参数化展开为平面得到的结果扭曲比直接参数化要小。

从上可以看出，三维模型的参数化并不直接作用于任意的三维模型，每个三维模型需要根据自己的拓扑属性和几何特点进行分割处理，得到近似于平面的碟形拓扑，再在碟形拓扑的三维模型上进行参数化，把碟形拓扑三维模型展开到平面上，从而得到三维模型和二维平面的一一映射结果。



（a）三维模型（碟形有边界）　（b）直接参数化

（c）分割为两部分　（d）第一部分参数化　（e）第二部分参数化

图 4-8　三个半球三维模型的切割

4.3　参数化算法分类

1. 扭曲度量分类

参数化算法有很多种，这些参数化算法根据不同的标准进行分类。对于碟形网格，也就是简单有边界的二维流型（2 - Manifold）网格，有几十种平面映射参数化算法。可以把这些算法根据采用的扭曲度量分为四大类。

（1）完全不考虑扭曲度。

（2）最小化角度扭曲。

（3）最小化面积扭曲。

（4）最小化拉伸扭曲。

2. 边界分类

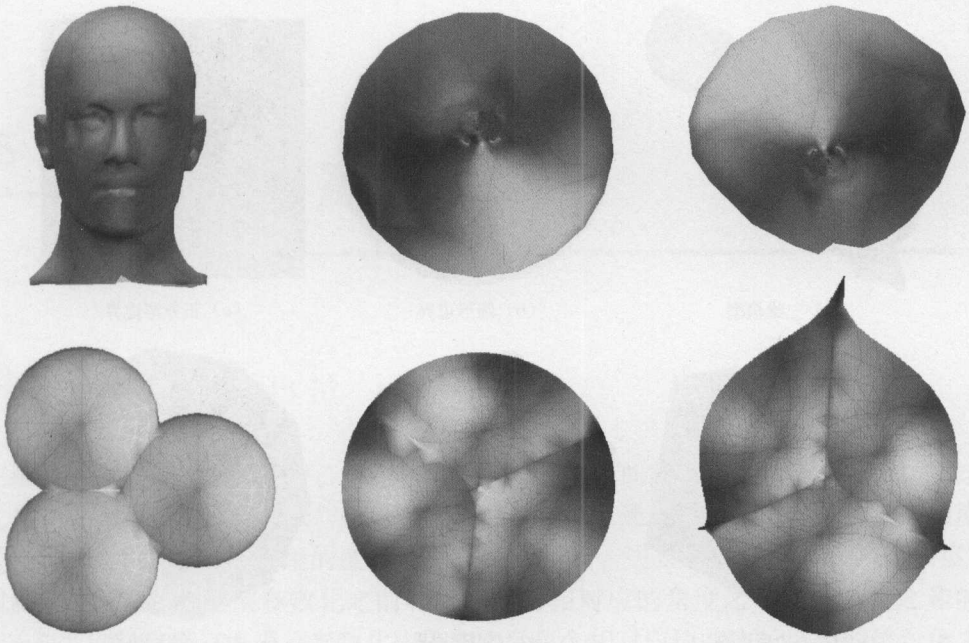
除了按照扭曲度来分类外，参数化算法还受到其他因素的影响。例如，按边界划分，有以下两大类。

（1）固定边界。

（2）不固定边界。

同一个模型固定边界和不固定边界的参数化结果分别如图 4-9 所示。

固定边界的参数化算法需要映射到的平面区域是固定边界（Fixed Boundary）的凸多边形（Convex）。而边界不固定（Free Boundary）的算法把边界的确定也作为算法的一部分。



（a）三维模型　　（b）固定边界参数化　　（c）不固定边界参数化

图 4-9　固定边界和不固定边界参数化算法效果

相对来说，固定边界算法简单、速度快。

固定边界的算法，需要先把边界映射到某个固定的二维形状，然后再映射其余非边界的顶点。如图 4-10 所示，把兔子头三维模型的边界先映射到一个圆形，然后再确定兔子头内部顶点的映射。

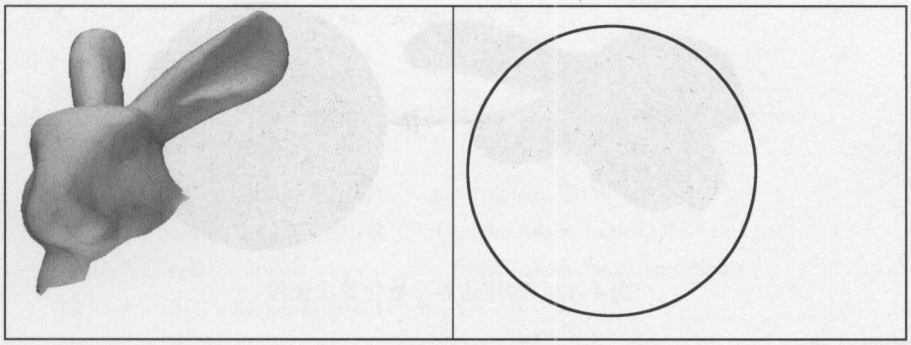


图 4-10　固定边界参数化算法效果

如图 4-11 所示，分别把兔子头三维模型参数化到圆形、正方形、五边形、六边形和十边形的边界。根据边界形状的不同，可以得到不同的展开平面形状。参数化的最终目的一般来说不是为了展开为平面，而是为了进行三维贴图，因此具体的平面形状并不重要。

具有边界的参数化映射过程如图 4-12 所示，第一行表示把三维模型兔子头的边界映射到一个圆，第二行表示把兔子模型上不是边界的顶点映射到圆的内部。

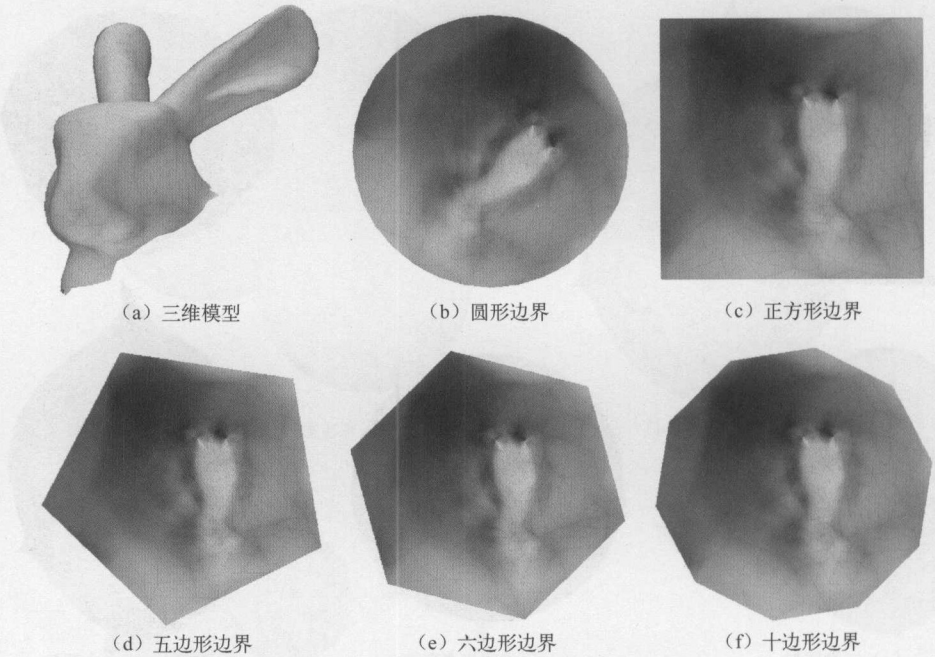


图 4-11　各种固定边界参数化算法效果

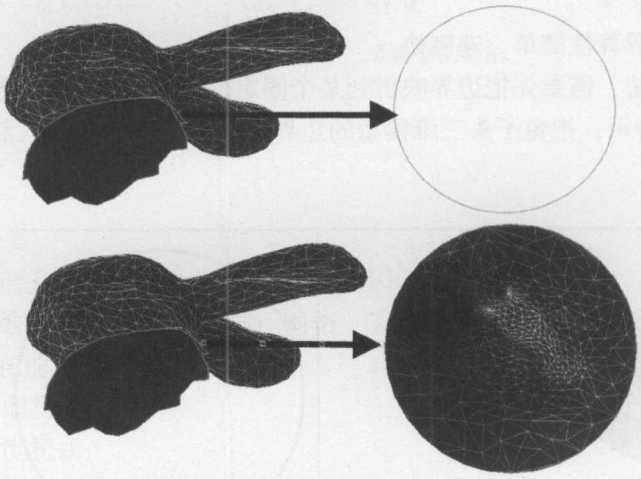


图 4-12　固定边界参数化算法流程

3. 其他分类

有些算法可以满足双射或者局部双射，有些算法只有在网格模型符合特定条件下才能满足双射。根据时间复杂度，还可以分为线性和非线性算法。线性算法通常速度快，但扭曲度高。如果一个参数法方法能够把本来是落在一个平面上的三维模型参数化后形状和原来一样，那么这种方法称为能够二维重现的参数化方法。如果参数化结果在局部上没有三角形翻转，以及两个三角形相交，在全局上没有边界扭度大于 2π 的情况，那么这个参数称为平坦化（Planarity）参数化。

第5章 扭曲度量

5.1 基本扭曲度量

三维模型在参数化的过程中，有各种各样的算法，因此需要定义一个度量来对比这些算法的优劣。这个度量也可以用来作为参数化算法的目标，通过设计数学系统来使扭曲度量最小，就得到参数化的结果。常用的度量有角度扭曲度量、边长扭曲度量、面积扭曲度量等。角度扭曲度量就是三维模型参数化之前和之后每对相对应的角度之差的绝对值之和的平均值。边长扭曲度量就是三维模型参数化之前和之后每个相对应边的边长之差的绝对值之和的平均值。面积扭曲度量就是三维模型参数化之前和之后每个三角形面积之差绝对值之和的平均值。这些度量不仅可以用来对比参数化算法的好坏，还可以指导设计参数化算法。某些参数化算法的设计方法就是通过优化问题来求解使这些度量最小的最优解。

各种扭曲度量公式如下所述。

1. 角度扭曲公式

$$\frac{1}{3F}\sum_{j}\sum_{i=1}^{3}|\theta_{j,i}-\phi_{j,i}|$$

式中，$\theta_{j,i}$和$\phi_{j,i}$分别表示参数化之前和之后的角度值。

代码如下。

```
public double[ ] ComputeDistortionAngles(TriMesh flatMesh,TriMesh backUpMesh)
{
    double[ ] paraAngles = TriMeshUtil. ComputeAngle(flatMesh);
    double[ ] backupAngles = TriMeshUtil. ComputeAngle(backUpMesh);
    double[ ] angleDistortions = new double[backUpMesh. HalfEdges. Count];
    for (int i =0; i <backupAngles. Length; i ++)
    {
        angleDistortions[i] = Math. Abs(paraAngles[i] - backupAngles[i]);
    }
    return angleDistortions;
}
```

2. 边长扭曲公式

$$\sum\left|\frac{|p_i-p_j|}{\sum|p_i-p_j|}-\frac{|u_i-u_j|}{\sum|u_i-u_j|}\right|$$

式中，p_i 表示三维模型参数化之前的边长；u_i 表示三维模型参数化后的边长。

代码如下。

```
public double[ ] ComputeDistortionEdges(TriMesh flatMesh,TriMesh backUpMesh)
{
    double[ ] flatLength = TriMeshUtil. ComputeEdgeLength(flatMesh);
    double[ ] backupLength = TriMeshUtil. ComputeEdgeLength(backUpMesh);
    double[ ] edgeDistortions = new double[flatMesh. Edges. Count];
    double paraedSum = 0;
    double backupSum = 0;
    for(int i = 0;i < flatLength. Length;i ++ )
    {
        paraedSum += flatLength[i];
        backupSum += backupLength[i];
    }
    for(int i = 0;i < flatLength. Length;i ++ )
    {
        double paraedValue = flatLength[i] / paraedSum;
        double backupValue = backupLength[i] / backupSum;
        edgeDistortions[i] = Math. Abs(paraedValue - backupValue);
    }
    return edgeDistortions;
}
```

3. 面积扭曲公式

$$\sum |A(T_j)/\sum A(T_j) - A(U_j)/\sum A(U_j)|$$

式中，$A(T_j)$表示三维模型参数化之前的三角形面的面积；$A(U_j)$表示三维模型参数化后的三角形面的面积。

代码如下。

```
public double[ ] ComputeDistortionAreas(TriMesh flatMesh,TriMesh backUpMesh)
{
    double[ ] flatAreas = TriMeshUtil. ComputeAreaFace(flatMesh);
    double[ ] backupAreas = TriMeshUtil. ComputeAreaFace(backUpMesh);
    double[ ] distortions = new double[flatMesh. Edges. Count];
    double flatSum = TriMeshUtil. ComputeAreaTotal(flatMesh);
    double backupSum = TriMeshUtil. ComputeAreaTotal(backUpMesh);
    for(int i = 0;i < flatAreas. Length;i ++ )
    {
        double paraedValue = flatAreas[i] / flatSum;
        double backupValue = backupAreas[i] / backupSum;
        distortions[i] = Math. Abs(paraedValue - backupValue);
    }
```

```
        return distortions;
    }
```

5.2　拉伸扭曲度量

5.2.1　仿射变换

除了上述比较简单、直观的扭曲度量之外，还有一种扭曲度量称为拉伸扭曲度量，这种扭曲度量能够更好地表示三维模型在参数化前后所发生的扭曲。拉伸扭曲度量把三维空间里的三维模型和展开后的二维平面上的三维模型上的三角形一一对应，对每一对三角形，根据这对三角形的顶点位置，计算仿射变换。这个仿射变换把三维空间的三角形变换为二维平面上的三角形。然后通过这个仿射变换的雅克比矩阵的特征值来度量三角形拉伸的程度。把每个三角形的拉伸求和就可以得到整个模型的拉伸程度。

1. 光滑曲面

在光滑的曲面上，对于三维空间里面和一个圆盘拓扑相等的一个曲面 $S \in R^3$，那么曲面的参数化表示形式为：

$$r(s,t) = [x(s,t), y(s,t), z(s,t)]$$

那么映射 $r(s,t)$ 的雅克比矩阵（Jacobian）为：

$$J = [\partial r/\partial s, \partial r/\partial t,]$$

Jacobian 矩阵决定了 $r(s,t)$ 映射的一阶几何性质，如面积，角度和长度扭曲。

如果用 $\Gamma(s,t)$ 和 $\gamma(s,t)$ 表示 Jacobian 的最大和最小的奇异值，那么

$$\mathrm{d}l^2 = E(s,t)\,\mathrm{d}s^2 + 2F(s,t)\,\mathrm{d}s\mathrm{d}t + G(s,t)\,\mathrm{d}t^2 \tag{5-1}$$

其中：

$$E = r_s^2, F = r_s \cdot r_t, G = r_t^2 \tag{5-2}$$

那么 Γ^2 和 γ^2 是 $J^{\mathrm{T}}J = \begin{bmatrix} E & F \\ F & G \end{bmatrix}$ 的特征值。假如 $\Gamma(s,t) = \gamma(s,t)$，那么映射是保角映射。

2. 离散模型

对于离散的三维模型来说，假设三维模型上的任意一个三角形 T 的三个顶点坐标三维表示为 q_1, q_2, q_3，这个三角形 T 参数化后的三个顶点坐标二维表示为 p_1, p_2, p_3，其中 $p_i = (s_i, t_i)$。二维三角形到三维三角形的映射构成一个仿射（Affine）。那么把二维三角形内部的一个顶点 p 映射到三维 q 表示为：

$$S(p) = S(s,t) = q$$

也就是：

$$S(p) = \frac{\langle p, p_2, p_3\rangle q_1 + \langle p, p_3, p_1\rangle q_2 + \langle p, p_1, p_2\rangle q_3}{\langle p_1, p_2, p_3\rangle}$$

其中 $\langle a,b,c\rangle$ 表示三角形面积。对上述仿射变换求偏导数得到：

$$S_s = \partial S/\partial s = (q_1(t_2 - t_3) + q_2(t_3 - t_1) + q_3(t_1 - t_2))/(2A)$$

$$S_t = \partial S/\partial t = (q_1(s_3 - s_2) + q_2(s_1 - s_3) + q_3(s_2 - s_1))/(2A)$$

$$A = \langle p_1, p_2 p_3\rangle = ((s_2 - s_1)(t_3 - t_1) - (s_3 - s_1)(t_2 - t_1))/2$$

3. 雅克比矩阵

上述两个偏导数可以构成一个 3×2 的雅克比矩阵。这个雅克比（Jacobian）矩阵[S_s, S_t]的最大和最小奇异（*Singular*）值为：

$$\Gamma=\sqrt{((a+c)+\sqrt{(a-c)^2+4b^2}/2}$$

$$\gamma=\sqrt{((a+c)-\sqrt{(a-c)^2+4b^2})/2}$$

式中，$a=S_s\cdot S_s$；$b=S_s\cdot S_t$；$c=S_t\cdot S_t$。

奇异值 Γ，γ 表示把二维单位向量映射到三维的最大和最小的长度，也就是最大和最小的局部拉伸。

当 $\frac{\Gamma}{\gamma}=1$ 时，映射前后角度保持不变。

当 $\Gamma\gamma=1$ 时，映射前后面积保持不变。

当 $\Gamma=\gamma=1$ 时，映射前后长度保持不变。

4. 拉伸范式

1）单个三角形

对于一个三角形，可以定义如下几种度量。

第一种是平均根二范式，公式为：

$$L^2(T)=\sqrt{(\Gamma^2+\gamma^2)/2}=\sqrt{(a+c)/2}$$

式中，$L^2(T)$ 表示在各个方向的平均（Root - mean - square）拉伸。

第二种是最大范式，公式为：

$$L^\infty(T)=\Gamma$$

式中，$L^\infty(T)$表示最大的拉伸，也就是最坏的情况。

第三种是综合考虑拉伸和收缩，公式如下：

$$D(T,T')=\max\{\gamma_{\max},1/\gamma_{\min}\}$$

根据放射变换得到的拉伸和扭曲，除了上面两个方法定义度量外。由于参数化扭曲的度量上拉长和收缩是一样重要的，也就是不能只考虑最大的拉伸，还需要考虑最小的收缩，把拉伸和收缩综合到一起，可以定义第三个扭曲度量。

第四种扭曲度量是类保角度（Quasi - Conformal）扭曲度量。定义为映射的雅克比（Jacobian）矩阵的最大和最小特征值的比率。类保角扭曲度量最小值为 1。

2）整个模型

上面计算的是单个三角形的拉伸度量，那么相应的整个三维模型的拉伸度量可以定义为：

$$L^2(M)=\sqrt{\sum_{T_i\in M}(L^2(T_i))^2A'(T_i)/\sum_{T_i\in M}A'(T_i)}$$

$$L^\infty(M)=\max_{T_i\in M}L^\infty(T_i)$$

式中，$A'(T_i)$是三维上三角形的面积。

5.2.2 拉伸扭曲度量代码

扭曲度量的实现主要是计算雅克比矩阵的奇异值。输入是三维模型和参数化的二维模

型，两个模型的顶点一一对应。扭曲度量是计算在一对三角形上，因此每个三角形上的奇异值计算出来之后，整个模型的扭曲度量就得到了。

1. 单个三角形扭曲度量

第一步：得到三角形参数化后二维顶点坐标。

```
double s1 = posFlat[0].x;
double s2 = posFlat[1].x;
double s3 = posFlat[2].x;

double t1 = posFlat[0].y;
double t2 = posFlat[1].y;
double t3 = posFlat[2].y;
```

第二步：计算三角形面积。

```
double area = ((s2 - s1) * (t3 - t1) - (s3 - s1) * (t2 - t1))/2;
```

第三步：计算中间变量。

```
double dT23 = t2 - t3;
double dT31 = t3 - t1;
double dT12 = t1 - t2;

double dS32 = s3 - s2;
double dS13 = s1 - s3;
double dS21 = s2 - s1;

Vector3D pos3D1 = origFace.GetVertex(0).Traits.Position;
Vector3D pos3D2 = origFace.GetVertex(1).Traits.Position;
Vector3D pos3D3 = origFace.GetVertex(2).Traits.Position;
```

第四步：计算映射函数的导数。

```
double dSsx = (pos3D1.x * dT23 +
               pos3D2.x * dT31 +
               pos3D3.x * dT12)/(2 * area);

double dSsy = (pos3D1.y * dT23 +
               pos3D2.y * dT31 +
               pos3D3.y * dT12)/(2 * area);

double dSsz = (pos3D1.z *  dT23 +
               pos3D2.z * dT31 +
               pos3D3.z * dT12)/(2 * area);

double dStx = (pos3D1.x * dS32 +
```

```
                pos3D2.x * dS13 +
                pos3D3.x * dS21)/(2 * area);

double dSty = (pos3D1.y * dS32 +
                pos3D2.y * dS13 +
                pos3D3.y * dS21)/(2 * area);

double dStz = (pos3D1.z * dS32 +
                pos3D2.z * dS13 +
                pos3D3.z * dS21)/(2 * area);
```

第五步：计算雅克比矩阵公式中的 a、b、c 变量。

```
double a = dSsx * dSsx + dSsy * dSsy + dSsz * dSsz;
double b = dSsx * dStx + dSsy * dSty + dSsz * dStz;
double c = dStx * dStx + dSty * dSty + dStz * dStz;
```

第六步：计算最大，最小奇异值。

```
double temp = Math.Sqrt((a - c) * (a - c) + 4 * b * b);
double ac = a + c;

double maxSingularity = Math.Sqrt(0.5 * (ac + temp));
double minSingularity = Math.Sqrt(0.5 * (ac - temp));
```

第七步：分别计算基于雅克比矩阵的扭曲度量。

```
stretch.RmsL2 = Math.Sqrt(ac/2);
stretch.MaxL8 = maxSingularity;
stretch.Min = minSingularity;

if(stretch.MaxL8 > 1/stretch.Min)
{
    stretch.Distortion = stretch.MaxL8;
}
else
{
    stretch.Distortion = 1/stretch.Min;
}
double origalArea = TriMeshUtil.ComputeAreaFace(origFace);
stretch.QuasiConformal = (stretch.MaxL8/stretch.Min) * origalArea;
```

整个函数代码如下。

```
public StretchTriangle  ComputeStretchTriangle(TriMesh.Face origFace,
                                                TriMesh.Face flatFace)
```

```
{
    StretchTriangle stretch = new StretchTriangle( );
    if( TriMeshUtil. ComputeAreaFace( flatFace) = =0)
    {
        stretch. Status = " UnDefined" ;
        return stretch;
    }
    List < Vector2D > posFlat = FlatFace( flatFace) ;

    double s1 = posFlat[ 0] . x;
    double s2 = posFlat[ 1] . x;
    double s3 = posFlat[ 2] . x;

    double t1 = posFlat[ 0] . y;
    double t2 = posFlat[ 1] . y;
    double t3 = posFlat[ 2] . y;

    double area = ( ( s2 - s1) * ( t3 - t1) - ( s3 - s1) * ( t2 - t1) )/2;

    double dT23 = t2 - t3;
    double dT31 = t3 - t1;
    double dT12 = t1 - t2;

    double dS32 = s3 - s2;
    double dS13 = s1 - s3;
    double dS21 = s2 - s1;

    Vector3D pos3D1 = origFace. GetVertex( 0) . Traits. Position;
    Vector3D pos3D2 = origFace. GetVertex( 1) . Traits. Position;
    Vector3D pos3D3 = origFace. GetVertex( 2) . Traits. Position;


    double dSsx = ( pos3D1. x * dT23 +
                    pos3D2. x * dT31 +
                    pos3D3. x * dT12) /( 2 * area) ;

    double dSsy = ( pos3D1. y * dT23 +
                    pos3D2. y * dT31 +
                    pos3D3. y * dT12) /( 2 * area) ;

    double dSsz = ( pos3D1. z *  dT23 +
                    pos3D2. z * dT31 +
```

```
                pos3D3.z * dT12)/(2 * area);

            double dStx = (pos3D1.x * dS32 +
                pos3D2.x * dS13 +
                pos3D3.x * dS21)/(2 * area);

            double dSty = (pos3D1.y * dS32 +
                pos3D2.y * dS13 +
                pos3D3.y * dS21)/(2 * area);

            double dStz = (pos3D1.z * dS32 +
                pos3D2.z * dS13 +
                pos3D3.z * dS21)/(2 * area);

            double a = dSsx * dSsx + dSsy * dSsy + dSsz * dSsz;
            double b = dSsx * dStx + dSsy * dSty + dSsz * dStz;
            double c = dStx * dStx + dSty * dSty + dStz * dStz;

            double temp = Math.Sqrt((a - c) * (a - c) + 4 * b * b);
            double ac = a + c;

            double maxSingularity = Math.Sqrt(0.5 * (ac + temp));
            double minSingularity = Math.Sqrt(0.5 * (ac - temp));

            stretch.RmsL2 = Math.Sqrt(ac/2);
            stretch.MaxL8 = maxSingularity;
            stretch.Min = minSingularity;

            if(stretch.MaxL8 > 1/stretch.Min)
            {
                stretch.Distortion = stretch.MaxL8;
            }
            else
            {
                stretch.Distortion = 1/stretch.Min;
            }
            double origalArea = TriMeshUtil.ComputeAreaFace(origFace);
            stretch.QuasiConformal = (stretch.MaxL8/stretch.Min) * origalArea;
            return stretch;
        }
```

2. 整个三维模型扭曲度量

在得到单个三角形的扭曲度量之后，整个三维模型的扭曲度量代码如下。

第一种扭曲度量。

```
public double[ ] ComputeStretchRmsL2(TriMesh orig,TriMesh flat)
{
    double[ ] rmsStretch = new double[orig. Faces. Count];
    for(int i =0;i < orig. Faces. Count;i ++)
    {
        StretchTriangle strech = ComputeStretchTriangle(orig. Faces[i],
                                                         flat. Faces[i]);
        rmsStretch[i] = strech. RmsL2;
    }
    return rmsStretch;
}
```

第二种扭曲度量。

```
public double[ ] ComputeStretchMaxL8(TriMesh orig,TriMesh flat)
{
    double[ ] maxStetch = new double[orig. Faces. Count];
    for(int i =0;i < orig. Faces. Count;i ++)
    {
        StretchTriangle strech = ComputeStretchTriangle(orig. Faces[i],
                                                         flat. Faces[i] );
        maxStetch[i] = strech. MaxL8;
    }
    return maxStetch;
}
```

第三种扭曲度量。

```
public double[ ] ComputeDistortion(TriMesh orig,TriMesh flat)
{
    double[ ] distortion = new double[orig. Faces. Count];
    for(int i =0;i < orig. Faces. Count;i ++)
    {
        StretchTriangle strech = ComputeStretchTriangle(orig. Faces[i],
                                                         flat. Faces[i]);
        distortion[i] = strech. Distortion;

    }
    return distortion;
}
```

第四种扭曲度量。

```
public double[ ] ComputeQuasiConformal(TriMesh orig,TriMesh flat)
{
    double[ ] qc = new double[orig. Faces. Count];
    for(int i =0;i < orig. Faces. Count;i ++ )
    {
        StretchTriangle strech = ComputeStretchTriangle(orig. Faces[i],
                                                        flat. Faces[i]);

        qc[i] = strech. QuasiConformal;
    }
    return qc;
}
```

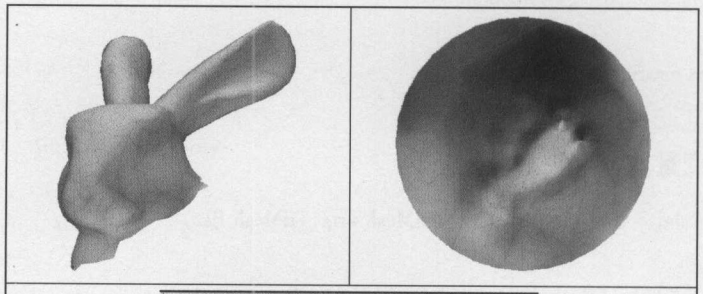
5.3 度量实验

1. 数值展示

下面的实验展示了几个三维模型经过特定的某个参数化算法展开后，各种度量的结果。每个实验分别展示了三维模型和参数化后的二维模型，以及各种度量的计算数值结果。

1）实验一

如图 5-1 所示，兔子头三维模型参数化算法的展开结果和各种扭曲度量的数值，每种扭曲度量的数值都不一样。对于不同的参数化算法，可以根据某个扭曲度量来判定参数化效果的好坏。

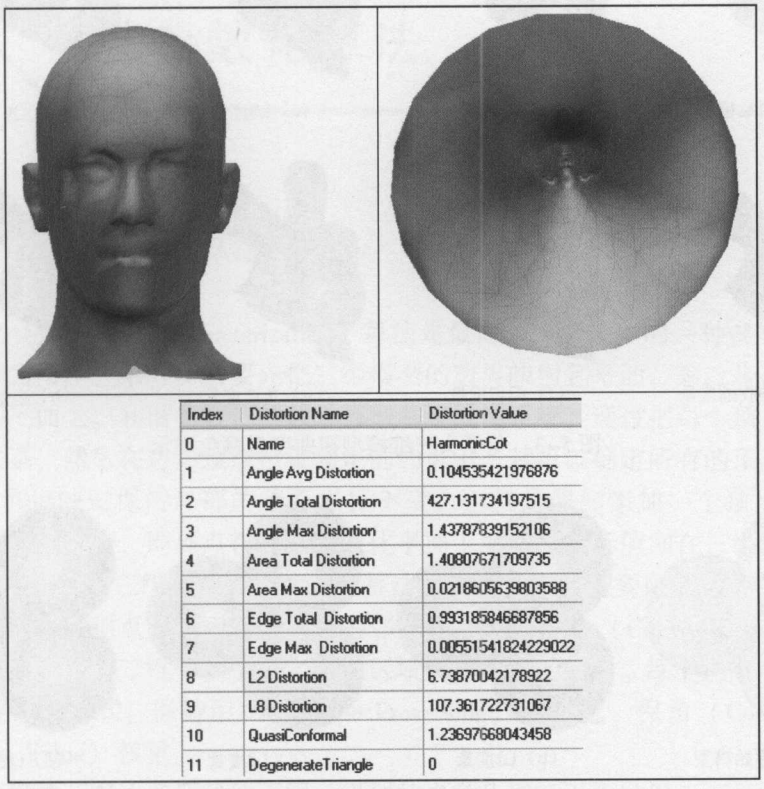


Index	Distortion Name	Distortion Value
0	Name	HarmonicCot
1	Angle Avg Distortion	0.141092721059909
2	Angle Total Distortion	617.421747358161
3	Angle Max Distortion	1.64799156291599
4	Area Total Distortion	1.15038678169621
5	Area Max Distortion	0.00919286267988662
6	Edge Total Distortion	0.855928089780375
7	Edge Max Distortion	0.00269266754679297
8	L2 Distortion	271.87434765985
9	L8 Distortion	2010.53740289741
10	QuasiConformal	1.52294107135021
11	DegenerateTriangle	0

图 5-1　兔子头参数化扭曲度量

2）实验二

如图5-2所示，人头三维模型参数化算法的扭曲度量结果，从中可以看出，人头三维模型在展开后鼻子部位的细节还保存，但边界扭曲就比较大。



Index	Distortion Name	Distortion Value
0	Name	HarmonicCot
1	Angle Avg Distortion	0.104535421976876
2	Angle Total Distortion	427.131734197515
3	Angle Max Distortion	1.43787839152106
4	Area Total Distortion	1.40807671709735
5	Area Max Distortion	0.0218605639803588
6	Edge Total Distortion	0.993185846687856
7	Edge Max Distortion	0.00551541824229022
8	L2 Distortion	6.73870042178922
9	L8 Distortion	107.361722731067
10	QuasiConformal	1.23697668043458
11	DegenerateTriangle	0

图5-2　人头参数化扭曲度量

2. 颜色展示

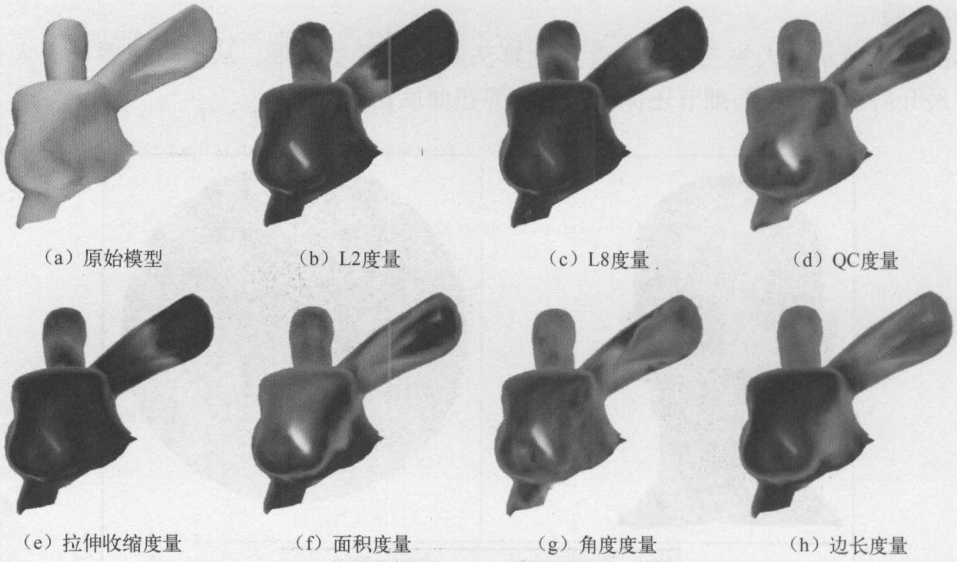
度量数值展示只是显示了整个三维模型的总体度量，很难看出来具体三维模型哪个部位扭曲比较大，哪个部位扭曲比较小。通过把三维模型每个三角形扭曲的数值映射到颜色上，可以发现扭曲最大的部位。红色部分表示扭曲最大的地方，蓝色部分表示扭曲比较小的地方。从下面两组实验可以看出，虽然各种度量的结果不同，但在三维模型弯曲比较大的部位，扭曲最大，而在三维模型弯曲比较小，近似平满的部位扭曲最小。例如，兔子耳朵部位扭曲比较大，而兔子面部由于近似平面，因此扭曲比较小。

1）实验一

如图5-3所示，兔子头三维模型参数化后的各种扭曲度量的颜色展示，其中红色部分表示扭曲比较大的地方，蓝色部分表示扭曲比较小的地方。可以看出，扭曲比较大的地方都分布在兔子耳朵等弯曲的部位，而兔子面部比较平缓的，近似于平面的部位扭曲比较小。

2）实验二

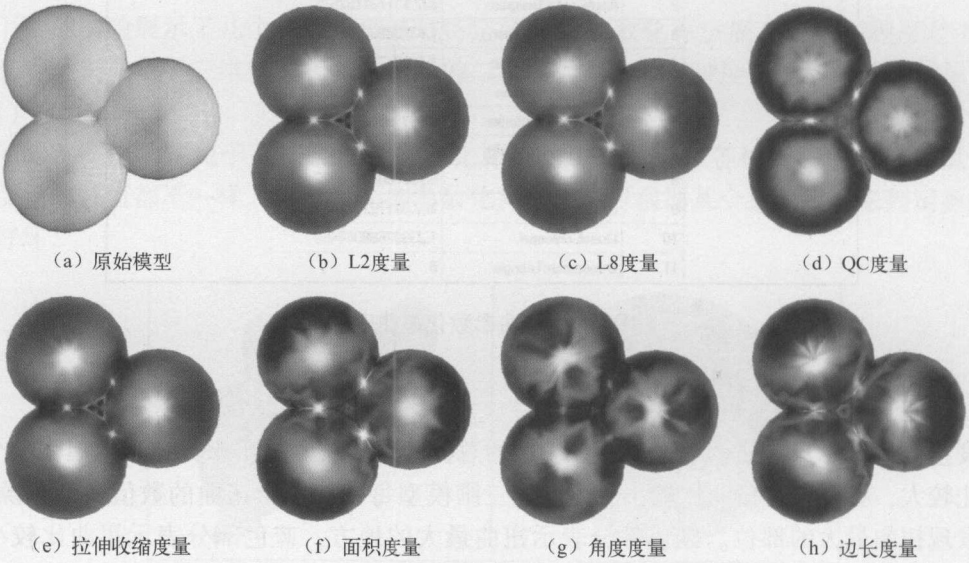
如图5-4所示，三个半球三维模型参数化后的各种扭曲度量颜色展示。虽然各种度量的颜色展示不一样，但都是在弯曲的地方扭曲比较大，平缓的地方扭曲较小。



（a）原始模型　（b）L2度量　（c）L8度量　（d）QC度量

（e）拉伸收缩度量　（f）面积度量　（g）角度度量　（h）边长度量

图 5-3　兔子头三维模型扭曲度量颜色展示



（a）原始模型　（b）L2度量　（c）L8度量　（d）QC度量

（e）拉伸收缩度量　（f）面积度量　（g）角度度量　（h）边长度量

图 5-4　三个半球三维模型扭曲度量颜色展示

第6章 和谐参数化算法

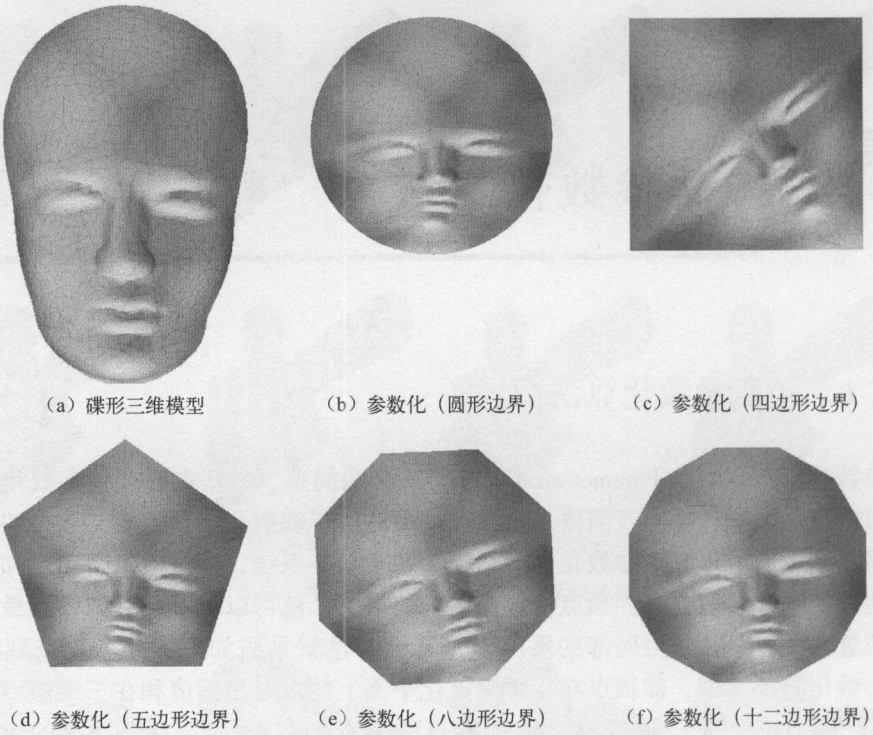
6.1 和谐参数化算法概述

和谐参数化（Harmonic Parameterization）算法是最简单、最基本的一种参数化方法。这种参数化方法分为两步：第一步需要先把三维模型的边界映射到平面；第二步再映射不是边界的内部顶点到平面上。和谐参数化的数学模型基于线性系统，通过把每个顶点和周围顶点建立一个线性关系，然后在边界顶点位置固定的约束下，就可以确定所有的顶点最终在平面上的展开位置。和谐映射保持局部的重心坐标不变，也就是假如添加一个顶点到三角形内部，那么参数化结果不变，即顶点在二维参数化平面上的重心坐标值和在三维模型上的重心坐标值一样。碟形拓扑的三维模型和谐参数化算法在边界固定时需要限制边界的形状为二维凸多边形，内部的每个顶点必须表示为和周围邻居顶点的凸组合（Convex Combination），这样才可以得到三维模型和二维平面上凸多边形区域内一对一的映射。每个顶点和邻居顶点的关系通常有三种权重方式：保持形状（Shape Preserving）的权重、保角（Conformal）的权重和中值（Mean Value）权重。

根据边界映射的凸多边形形状的不同，得到的参数化展开平面结果也不一样。如图6-1所示的一个人脸的三维模型在圆形边界、四边形、五边形、八边形、十二边形边界上的展开结果。可以看出人脸的眼睛、鼻子、嘴巴细节都保持，但边界形状不一样。

和谐参数化（Harmonic Map）的构造中参数之间的关系是间接耦合的，因此需要先确定三维模型的边界和到平面上的边界的映射。和谐参数化算法的缺点是需要一个凸多边形的边界，但很多三维模型的边界不是凸多边形的，因此参数化后会产生很大的扭曲。例如，对一个本来就落在一个平面上但边界不是凸多边形的三维模型来说，经过参数化后，还应该和原来的形状一样，但用这种方法会产生非常大的扭曲。和谐参数化的其他缺点是：不保角，不保面积，只能处理碟形拓扑结构的模型，还可能出现三角形翻转情况。虽然和谐参数化算法没有针对特定的度量进行设计和优化，也就是无法保证某些扭曲度量能够得到最小值，但该方法简单直接、原理清楚，是所有三维模型参数化的基本参照算法。通过和谐参数化算法可以了解所有线性参数化算法的数学模型构建的基本思路，以及得到线性系统之后的数值求解方法。

和谐参数化定义：给定一个碟形的具有边界的三维网格模型 M，如果先把边界映射到一个封闭的平面曲线 C，那么通过一个和谐函数 $F:M\rightarrow R^2$，可以把这个三维网格模型映射到平面上。



（a）碟形三维模型　（b）参数化（圆形边界）　（c）参数化（四边形边界）

（d）参数化（五边形边界）　（e）参数化（八边形边界）　（f）参数化（十二边形边界）

图 6-1　不同边界形状的和谐参数化展开效果图

6.2　和谐参数化系统构造

6.2.1　系统构造过程

所有参数化算法的已知条件都是三维模型的顶点几何位置和拓扑连接关系，未知的是映射到平面上的顶点位置。把三维模型映射到二维平面有无数种可能的结果，只有加上额外的限制条件，才能得到唯一的结果。根据额外的限制条件的不同，得到的结果不一样。从而采用的数学模型和相应的算法也不一样。这些限制条件有的是需要展开后得到的角度尽可能保持不变，有的是展开后的面积尽可能不变，也有的是其他等限制条件。根据每种限制条件都可以构建一个数学系统，求解这个数学系统，就可以得到展开后的唯一结果。否则，不加入限制条件，一个三维模型展开到二维平面会有无数个可能性。根据已知条件和限制条件，求解位置的顶点位置的过程就是数学模型建立的过程。数学模型可能是线性的或非线性的。非线性的求解比较复杂，时间复杂度比较高，而线性的系统求解比较快速，常常需要把非线性的系统简化为线性的系统。参数化算法有非线性的和线性的系统，本书着重介绍线性系统的构建。线性系统需要求解线性方程组，构建的过程主要是构建方程组左边的矩阵和方程组右边的向量，而未知数就是平面上展开的顶点位置。根据每种参数化限制条件的不同，构建的数学模型也不同，也就是构建的矩阵和向量不同。但在得到矩阵和向量之后，求解线性方程组的方法都是一样的。

和谐参数化的数学系统构建过程如下所述。

（1）对于给定的函数，可以构建一个线性系统，此线性系统中所有内部顶点的拉普拉斯值都是零，边界点的值是事先映射的值。假设网格模型由 n 个顶点组成：

$$\{v_1,\cdots,v_m,v_{m+1},\cdots,v_n\}\subset R^3$$

式中，$\{v_1,\cdots,v_m\}$ 是边界顶点，$\{v_{m+1},\cdots,v_n\}$ 是内部节点。参数含义如图 6-2 所示。

（2）把边界顶点 v_i 映射到平面一个封闭曲线 c_i，也就是：

$$v_i\rightarrow c_i,1\leqslant i\leqslant m,c_i\in R^2$$

（3）对于内部顶点需要满足如下条件：

$$(Lv)_i=0,m+1\leqslant i\leqslant n$$

（4）用 P_i 表示平面上参数化映射后的位置，从而可以构成一个线性方程，如下：

$$\left(\begin{array}{ccc|c}1 & L & 0 & \\ \vdots & O & \vdots & 0\\ 0 & L & 1 & \\ \hline \multicolumn{3}{c|}{\boldsymbol{L}_{[1,m]\times[m+1,n]}} & \boldsymbol{L}_{[m+1,n]\times[m+1,n]}\end{array}\right)\begin{pmatrix}p_1\\ \vdots\\ p_m\\ p_{m+1}\\ \vdots\\ p_n\end{pmatrix}=\begin{pmatrix}c_1\\ \vdots\\ c_m\\ 0\\ \vdots\\ 0\end{pmatrix}$$

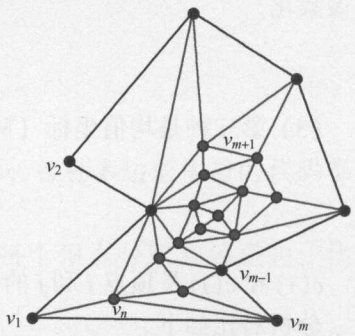


图 6-2　参数含义

（5）这个线性系统不是对称的，上面的系统可以化简为如下形式：

$$\boldsymbol{L}_{[1,m]\times[m+1,n]}\begin{pmatrix}c_1\\ \vdots\\ c_m\end{pmatrix}+\boldsymbol{L}_{[m+1,n]\times[m+1,n]}\begin{pmatrix}p_{m+1}\\ \vdots\\ p_n\end{pmatrix}=\begin{pmatrix}0\\ \vdots\\ 0\end{pmatrix}$$

（6）把系统重新整理后，得到如下对称的线性系统：

$$\boldsymbol{L}_{[m+1,n]\times[m+1,n]}\begin{pmatrix}p_{m+1}\\ \vdots\\ p_n\end{pmatrix}=-\boldsymbol{L}_{[1,m]\times[m+1,n]}\begin{pmatrix}c_1\\ \vdots\\ c_m\end{pmatrix}$$

（7）每个顶点参数化后的坐标是二维点 (u,v)，把整个模型所有顶点的 u 坐标和 v 坐标分别构成两个向量 $\boldsymbol{u}$ 和 $\boldsymbol{v}$，因此未知数是两个向量 $\boldsymbol{u}$ 和 $\boldsymbol{v}$。因此需要两个线性系统，对 $\boldsymbol{u}$ 和 $\boldsymbol{v}$ 单独进行求解，这两个线性系统是独立的，互不关联。

$$L_u=0$$
$$L_v=0$$

（8）线性系统的矩阵是个拉普拉斯矩阵，形式如下：

$$\boldsymbol{L}_{i,j}=\begin{cases}-\sum\limits_{k\neq 1}w_{i,k} & i=j\\ w_{ij} & (i,j)\in E\\ 0 & \text{其他}\end{cases}$$

6.2.2　和谐参数化中的权重

在和谐参数化构建的线性系统中的拉普拉斯矩阵里，w_{ij} 是权值，假如权值是正值，同时

上述矩阵 $\boldsymbol{L}$ 是对称矩阵，那么参数化后的结果是双射的。选择不同的权重得到不同的结果。通常有三种权重。

（1）所有权重都一样（Uniform），都是 1 的情况下，矩阵 $\boldsymbol{L}$ 是经典的图形拉普拉斯矩阵。

$$w_{ij} = 1$$

（2）第二种是和谐（Harmonic）权重，计算方法如下所述。在 6.2.3 节会详细介绍这种参数化。

$$w_{ij} = \frac{(\cot\alpha_{ij} + \cot\beta_{ij})}{2}$$

（3）第三种是均值坐标（Mean Value）权重方法。

$$w_{ij} = \frac{\tan\left(\frac{\gamma_{ij}}{2}\right) + \tan\left(\frac{\delta_{ij}}{2}\right)}{\|c(i) - c(j)\|}$$

$c(i)$ 和 $c(j)$ 是顶点 i 和 j 的位置三维坐标。其他的参数含义如图 6-3 所示。

分析对比如下。

（1）在平均权值的权重中，参数化的结果是双射的，但不保持原来网格的很多特性，也称为中心映射（Barycentric Map），这种映射结果是双射的。

（2）在和谐权值方法中，如果三角形有钝角，那么权值可能是负值，从而参数化结果不是双射的。但假如网格满足德劳内（Delaunay）条件，那么即使有钝角，最终的参数化结果也是双射的。

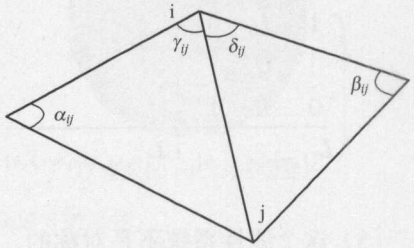


图 6-3　参数含义

（3）在均值权值方法中，虽然矩阵不是对称的，但参数化结果还是双射的。在大多数情况下，虽然相对于和谐方法来说，均值权值方法获得较小的角度变形，但对于有些模型的角度扭曲会更大。

6.3　和谐参数化代码

6.3.1　设置边界代码

和谐参数化是固定边界的算法，需要先设置边界。一般来说，有两种类型的边界：圆形和多边形。设置边界函数的输入是边界上的顶点，输出是这些顶点的二维坐标。下面是圆形和多边形边界形状的构造函数。

1. 圆形边界

圆形边界设置比较简单，根据边界顶点的数量，在圆形上平均取同样数量的点，代码如下。

```
public static Vector2D[ ] BuildCircle(TriMesh. Vertex[ ] vertices)
{
    List < Vector2D > list = new List < Vector2D > ( );
    double angle = Math. PI * 2/vertices. Length;
```

```
        for(int i=0;i<vertices.Length;i++)
        {
            double x=Math.Cos(angle*i);
            double y=-Math.Sin(angle*i);
            list.Add(new Vector2D(x,y));
        }
        return list.ToArray();
    }
```

2. 多边形边界

多边形边界函数可以根据边数的不同得到不同的多边形，各种多边形构造的代码都一样，边数可以作为参数进行输入。

（1）按照多边形边的个数和输入的边界顶点个数，把每个顶点分组到每条边，代码如下。

```
public static TriMesh.Vertex[][] Group(TriMesh.Vertex[] vertices,
                                       int edge)
{
    if(edge<3 || edge>vertices.Length)
    {
        throw new Exception("edge 不对");
    }
    List<TriMesh.Vertex[]>all=new List<HalfEdgeMesh.Vertex[]>();
    List<TriMesh.Vertex>edgeList=new List<HalfEdgeMesh.Vertex>();
    double step=(double)vertices.Length/edge;
    double sum=0.0000001;
    foreach(var v in vertices)
    {
        edgeList.Add(v);
        sum++;
        if(sum>=step)
        {
            sum-=step;
            all.Add(edgeList.ToArray());
            edgeList.Clear();
        }
    }
    return all.ToArray();
}
```

（2）把每条边的第一个顶点抽取出来作为一个集合，把这个集合的点映射到圆上，代码如下。

```
public static Vector2D[ ] BuildPolygon(TriMesh. Vertex[ ][ ] group)
{
    List < TriMesh. Vertex > vList = new List < HalfEdgeMesh. Vertex > ( ) ;
    foreach(var edge in group)
    {
        vList. Add(edge[0]) ;
    }
    Vector2D[ ] vPos = BuildCircle(vList. ToArray( )) ;
    return BuildPolygon(group,vPos) ;
}
```

（3）得到每条边第一个顶点的位置后，这条边其他顶点的位置计算如下。

```
public static Vector2D[ ] BuildPolygon(TriMesh. Vertex[ ][ ] group,
                                       Vector2D[ ] vPos)
{
    List < Vector2D > all = new List < Vector2D > ( ) ;
    for(int i = 0;i < group. Length;i ++ )
    {
        Vector2D cur = vPos[i] ;
        Vector2D next = vPos[(i + 1) % vPos. Length] ;
        all. Add(cur) ;
        for(int j = 1;j < group[i]. Length;j ++ )
        {
            all. Add(cur + (next - cur) * j/group[i]. Length) ;
        }
    }
    return all. ToArray( ) ;
}
```

6.3.2 线性系统代码

构造线性系统的代码就是生成矩阵和向量的过程。根据权重的不同，得到的矩阵和向量也不一样，下面分别对三种权重构造矩阵和向量。在得到矩阵和向量之后，未知数就可以通过求解线性方程的方法得到。

（1）建立三种权重的矩阵。

① 平均权重。

```
public SparseMatrix BuildMatrixCombinatorialGraphNormalized(TriMesh mesh)
{
    int n = mesh. Vertices. Count;
    SparseMatrix L =    BuildAdjacentMatrixVV(mesh) ;
    for(int i = 0;i < n;i ++ )
```

```
        {
            double sum = L. Rows[i]. Count;
            foreach(SparseMatrix. Element e in L. Rows[i])
            {
                L. AddValueTo(i,e. j,1/sum - 1);
            }
            L. AddValueTo(i,i, -1);
        }
        L. SortElement();
        return L;
    }
```

② 和谐权重。

```
    public SparseMatrix BuildLaplaceMatrixCotBasic(TriMesh mesh)
    {
        int n = mesh. Vertices. Count;
        SparseMatrix L = new SparseMatrix(n,n);
        for(int i = 0;i < mesh. Faces. Count;i ++)
        {
            int c1 = mesh. Faces[i]. GetVertex(0). Index;
            int c2 = mesh. Faces[i]. GetVertex(1). Index;
            int c3 = mesh. Faces[i]. GetVertex(2). Index;
            Vector3D v1 = mesh. Faces[i]. GetVertex(0). Traits. Position;
            Vector3D v2 = mesh. Faces[i]. GetVertex(1). Traits. Position;
            Vector3D v3 = mesh. Faces[i]. GetVertex(2). Traits. Position;
            double cot1 = (v2 - v1). Dot(v3 - v1)/(v2 - v1). Cross(v3 - v1). Length();
            double cot2 = (v3 - v2). Dot(v1 - v2)/(v3 - v2). Cross(v1 - v2). Length();
            double cot3 = (v1 - v3). Dot(v2 - v3)/(v1 - v3). Cross(v2 - v3). Length();
            L. AddValueTo(c1,c2,cot3);L. AddValueTo(c2,c1,cot3);
            L. AddValueTo(c2,c3,cot1);L. AddValueTo(c3,c2,cot1);
            L. AddValueTo(c3,c1,cot2);L. AddValueTo(c1,c3,cot2);
        }
        return L;
    }
    public SparseMatrix BuildMatrixCot(TriMesh mesh)
    {
        int n = mesh. Vertices. Count;
        SparseMatrix L =    BuildLaplaceMatrixCotBasic(mesh);
        for(int i = 0;i < n;i ++)
        {
            double sum = 0;
            foreach(SparseMatrix. Element e in L. Rows[i])
```

```
            {
                sum += e. value;
            }
            L. AddValueTo(i,i, -sum);
        }
        L. SortElement();
        return L;
    }
```

③ 均值权重。

```
public SparseMatrix BuildMatrixMeanValue(TriMesh mesh)
{
    int n = mesh. Vertices. Count;
    SparseMatrix L =    BuildAdjacentMatrixVV(mesh);
    SparseMatrix D = new SparseMatrix(n,n);
    foreach(TriMesh. HalfEdge halfedge in mesh. HalfEdges)
    {
        TriMesh. Vertex fromJ = halfedge. FromVertex;
        TriMesh. Vertex toI = halfedge. ToVertex;
        Vector3D jtoI = (toI. Traits. Position - fromJ. Traits. Position). Normalize();
        Vector3D alphaToI = (halfedge. Next. ToVertex. Traits. Position
                            - toI. Traits. Position). Normalize();
        Vector3D betaToI = (halfedge. Opposite. Previous. FromVertex. Traits. Position
                            - toI. Traits. Position). Normalize();
        double cosGamaIJ = jtoI. Dot(alphaToI)/(jtoI. Length() * alphaToI. Length());
        double cosThetaIJ = jtoI. Dot(betaToI)/(jtoI. Length() * betaToI. Length());
        double angelGama = Math. Acos(cosGamaIJ);
        double angelTheta = Math. Acos(cosThetaIJ);
        double wij = (Math. Tan(angelGama/2) + Math. Tan(angelTheta/2))
                    /(toI. Traits. Position - fromJ. Traits. Position). Length();
        int indexI = toI. Index;
        int indexJ = fromJ. Index;
        D. AddValueTo(indexI,indexJ,wij);
    }
    for(int i =0;i < n;i ++)
    {
        double sum =0;
        foreach(SparseMatrix. Element item in D. Rows[i])
        {
            sum += item. value;
```

```
            }
            D. AddValueTo(i,i, -sum);
        }
        return D;
    }
```

（2）拉普拉斯矩阵建立后，需要把边界顶点对应的行里的项设置为1，代码如下。

```
for(int i=0;i<boundary. UV. Boundary. Count;i++)
{
    int boundaryVertex = boundary. UV. Boundary[i]. Index;
    for(int j=0;j<n;j++)
    {
        LeftMatrix[boundaryVertex,j]=0;
    }
    LeftMatrix[boundaryVertex,boundaryVertex]=1;
}
```

（3）在得到边界映射的顶点位置之后，线性系统的右边向量就确定了，因为两个坐标分量独立求解，需要构建两个向量，代码如下。

```
public double[][] BuildRightB(ParaBoundary boundary)
{
    int m = Mesh. Vertices. Count;
    int n=2;
    double[][] RightB = new double[n][];
    for(int j=0;j<n;j++)
    {
        RightB[j] = new double[m];
    }
    for(int i=0;i<m;i++)
    {
        RightB[0][i]=0;
        RightB[1][i]=0;
    }
    for(int i=0;i<boundary. UV. Boundary. Count;i++)
    {
        TriMesh. Vertex vertex = boundary. UV. Boundary[i];
        RightB[0][vertex. Index] = boundary. UV. Pos[i]. x;
        RightB[1][vertex. Index] = boundary. UV. Pos[i]. y;
    }
```

```
        return RightB;
    }
```

（4）在构建好线性系统的左边矩阵和右边向量之后，就可以通过线性系统库进行求解，得到的值就是三维模型顶点参数化到二维后的坐标，代码如下。

```
public void Parameterlize(EnumParaHarmonicType type)
{
    ParaBoundary boundary = new ParaBoundary(Mesh);
    boundary. SetUpBoundary();
    SparseMatrix LeftMatrix = BuildMatrixLeft(type,boundary);
    double[][] RightB = BuildRightB(boundary);
    SolveLinearSystem(LeftMatrix,RightB);
    LeftMatrix = null;
    RightB = null;
    GC. Collect();
}
```

6.4 和谐参数化效果分析

如图 6-4 所示是各种不同形状的三维模型在三种权重下的参数化结果，从中可以得到如下结论。

（1）不同的权重，构建出来的二维平面不一样。

（2）尤其是点数比较少的模型，差别越明显。

（3）并且在靠近边界的地方，三角形的扭曲越大。因此固定边界的参数化算法，对于边界的扭曲没有有效的控制。

（4）虽然理论上和谐权值有面翻转的情况，但在实际中，即使在扭曲很大的情况下，也很少发生面翻转的结果。例如，球的三维模型，边界只是一个很小的洞，展开成平面之后扭曲很大，但也没有出现面翻转现象。

（5）假如三维模型已经是一个平面的三维模型，三种参数化后，原来顶点的位置仍会发生变化，但和谐参数化结果最能够保持原来的顶点位置不变，如图 6-4 中的第 6 行所示。

纹理贴图如下。

在三维模型参数化成平面后，就可以把二维图像贴到三维模型上了，如图 6-5 所示，当图片贴在二维参数化后的模型上时，效果很好，但返回到三维空间时，就发生了很大的扭曲，尤其是兔子的头部等地方。但是假如三维模型本身近似于一个平面，如图中面具的三维模型，贴上图之后，图片的扭曲就比较少。因此，和谐化的参数化算法在贴图应用中需要三维模型本身近似于一个平面。

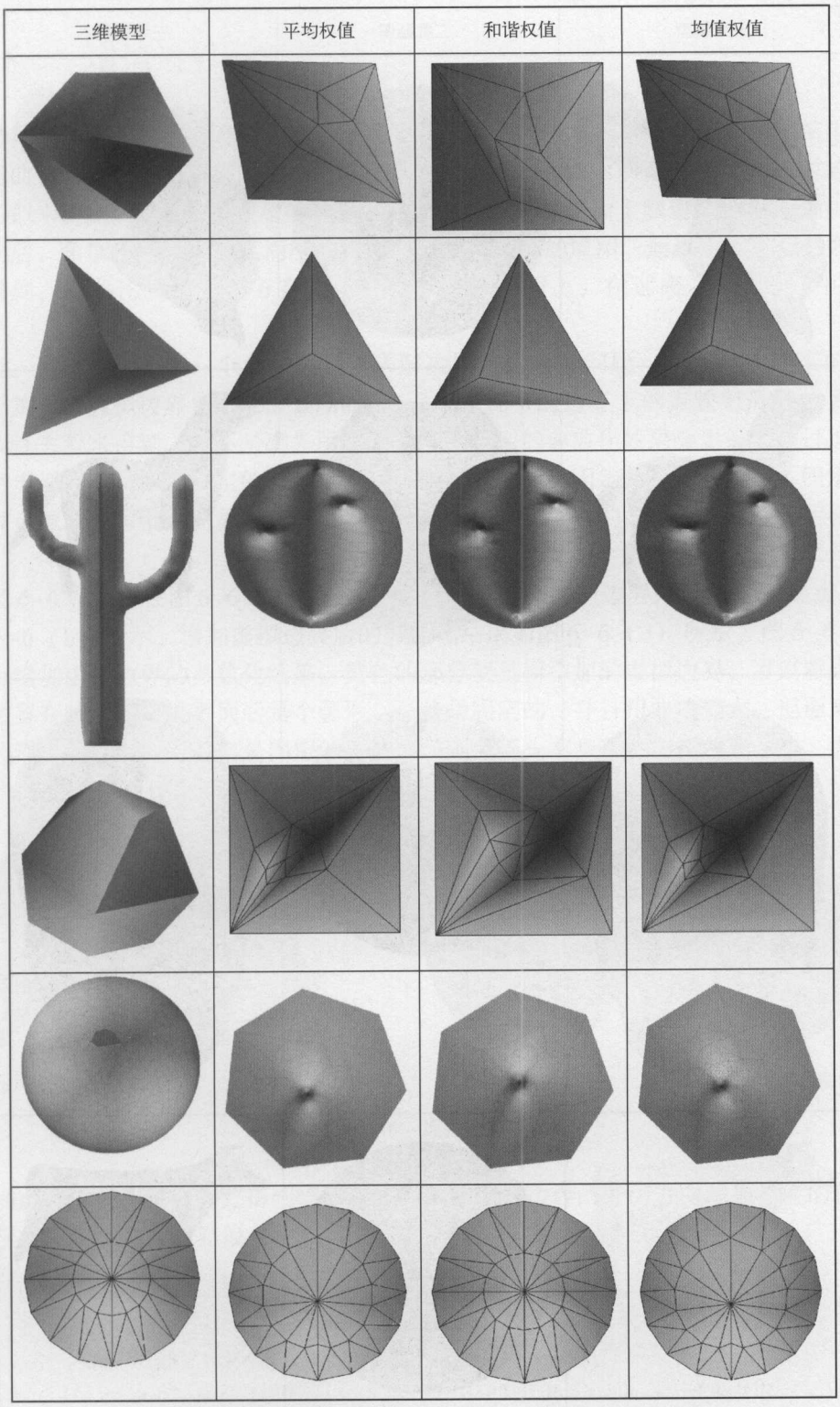


图 6-4　各种三维模型和谐参数化展开效果图

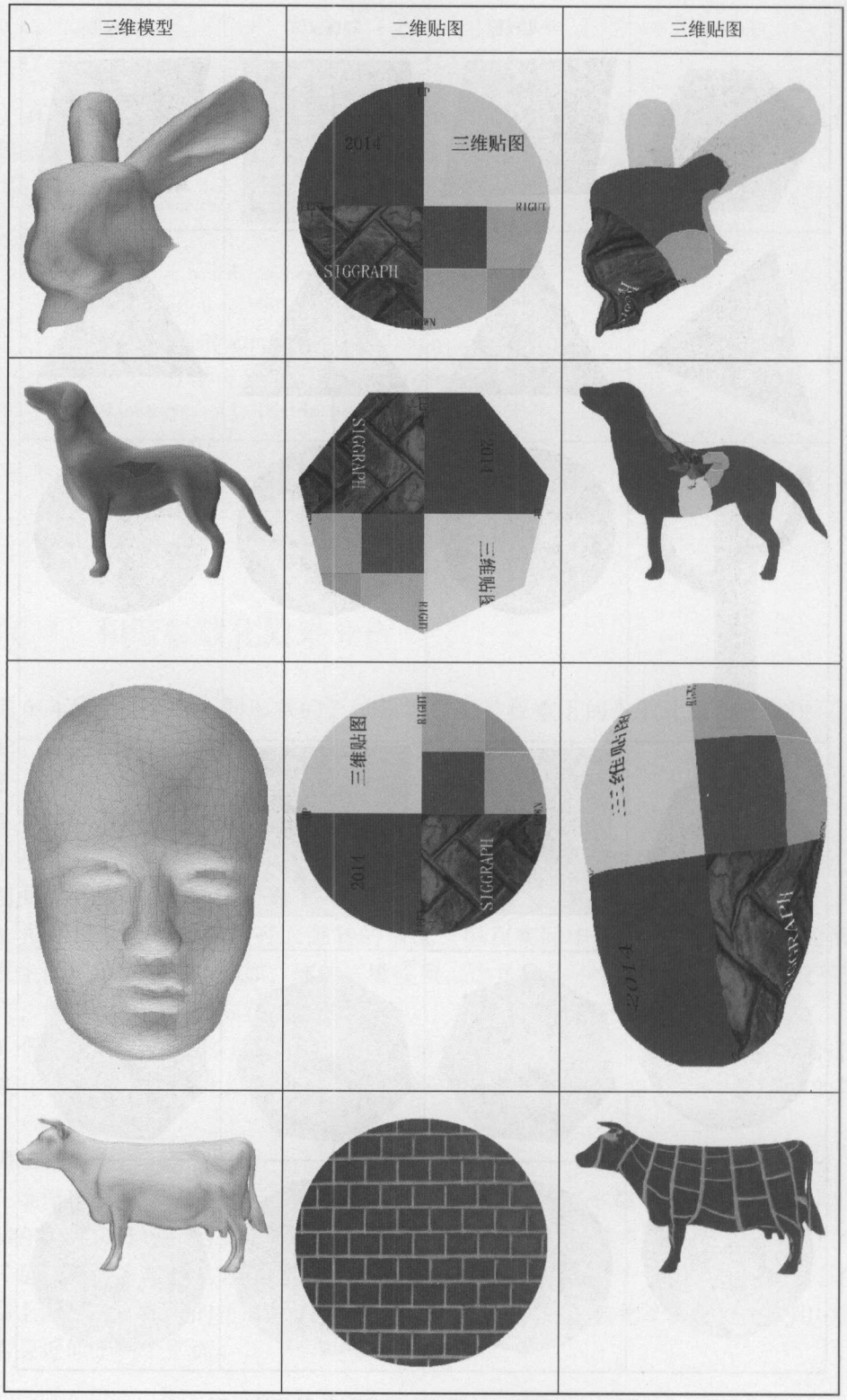


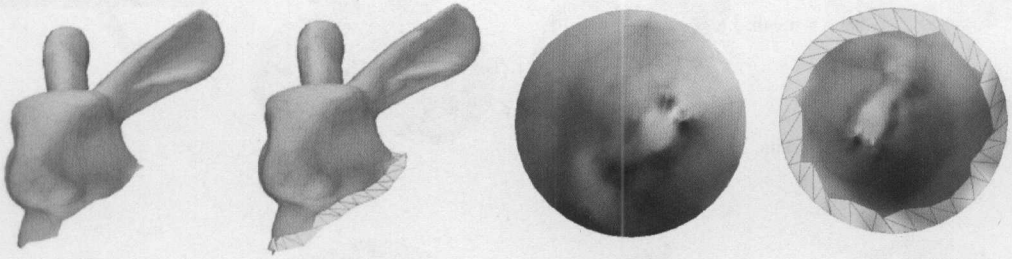
图 6-5　各种三维模型和谐参数纹理贴图

6.5　虚拟边界

和谐参数化方法简单，容易实现，只需要解一个线性方程组。但和谐参数化结果会发生很大的扭曲，尤其是在边界附近的三角形面。角度扭曲的程度由网格的边界和映射的边界形状是否相似来决定。假如网格模型的边界不是凸多边形或和平面上映射的边界差别很大，那么参数化后，角度扭曲很大，从而限制了和谐参数化算法的应用。通过观察和谐参数化的结果可以看到，扭曲比较大的地方都发生在靠近边界的面上，在远离边界面上，扭曲都比较小。

因此，可以通过引入一个虚拟边界，来解决固定边界的问题，也就是在原来的边界外面，再添加一层虚拟边界（Virtual Boundary），从而现有的边界上的点映射后的位置就可以通过解线性方程来得到，而不是事先固定的。这样得到的参数化效果会更好。经过这样的处理，虽然虚拟边界仍旧是固定的，但原始模型的边界变为不固定的，从而达到不固定边界的效果。并且原始模型上的面都远离虚拟边界，因此原始模型上的面在和谐参数化后扭曲就比较小。

如图 6-6 所示，在图 6-6（a）所示的兔子头三维模型的边界外面增加一层虚拟边界，变为图 6-6（b）所示，增加虚拟边界后的参数化结果如图 6-6（d）所示，图 6-6（c）所示是没有增加虚拟边界的参数化结果。黄色的部分就是新添加的虚拟边界，可以看出，黄色部分的边界在映射到二维平面后是个圆形，形状是固定的，并且扭曲比较大，但原来兔子头本身的边界不是圆形，也就不是固定的，从而得到兔子头本身边界的扭曲比较小。大部分扭曲都集中在虚拟边界上了。



（a）原始模型　（b）增加虚拟边界　（c）原始模型参数化　（d）虚拟边界参数化

图 6-6　虚拟边界

增加虚拟边界的算法首先得到模型的边界，然后在边界的外围方向添加相应的顶点，并把这些顶点和原始模型边界顶点相连，生成新的三角形。完整的代码如下。

```
public static void BoundaryExpand(TriMesh mesh)
{
    double length = TriMeshUtil. ComputeEdgeAvgLength(mesh);
    List < List < TriMesh. HalfEdge >> holes = TriMeshUtil. RetrieveBoundaryEdgeAll(mesh);
    foreach(var hole in holes)
    {
```

```
        TriMesh.Vertex[] arr = new HalfEdgeMesh.Vertex[hole.Count];
        for(int i = 0;i < hole.Count;i ++)
        {
            Vector3D normal = Vector3D.UnitX;
            if(hole[i].Opposite.Face != null)
            {
                normal = TriMeshUtil.ComputeNormalFace(hole[i].Opposite.Face);
            }
            Vector3D toPos = hole[i].ToVertex.Traits.Position;
            Vector3D fromPos = hole[i].FromVertex.Traits.Position;
            Vector3D hfDir = toPos - fromPos;
            Vector3D hfMid = (toPos + fromPos)/2;
            Vector3D pos = hfMid + normal.Cross(hfDir).Normalize() * length;
            arr[i] = mesh.Vertices.Add(new VertexTraits(pos));
        }
        for(int i = 0;i < hole.Count;i ++)
        {
            int next = (i + 1) % hole.Count;
            TriMesh.Face face = mesh.Faces.Add(arr[i],
                                               hole[i].ToVertex,
                                               hole[next].ToVertex);
            face.Traits.SelectedFlag = 1;
            face = mesh.Faces.Add(arr[i],
                                  hole[next].ToVertex,
                                  arr[next]);
            face.Traits.SelectedFlag = 1;
        }
        }
    }
```

1. 效果图

通过对增加多层虚拟边界，可以使原始模型的边界更自由，如图 6-7 所示，分别是一个三维模型在没有虚拟边界、一层虚拟边界、二层虚拟边界情况下的参数化结果。从中可以看出，如果原始三维模型近似于一个平面，那么参数化后的扭曲较少，增加虚拟边界后的扭曲减少得也比较少。例如，第一行的面具模型，增加虚拟边界后和没有虚拟边界的区别并不大。假如原来的三维模型弯曲比较大，那么增加虚拟边界就可以很大地减少参数化展开后的扭曲。例如，第二行人头模型和第三行的三个半球模型在增加虚拟边界后，得到的参数化结果和不固定边界的结果类似，极大地减少了面的扭曲。

2. 纹理贴图

如图 6-8 所示是各种不同形状三维模型经过虚拟边界参数化后的贴图结果。

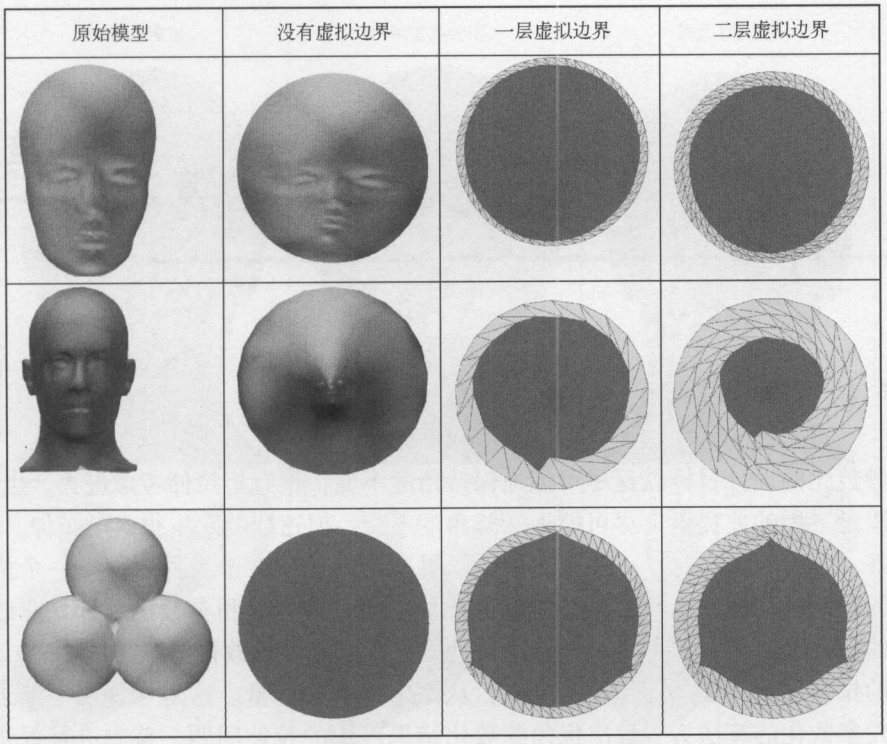


图 6-7　多层虚拟边界参数化

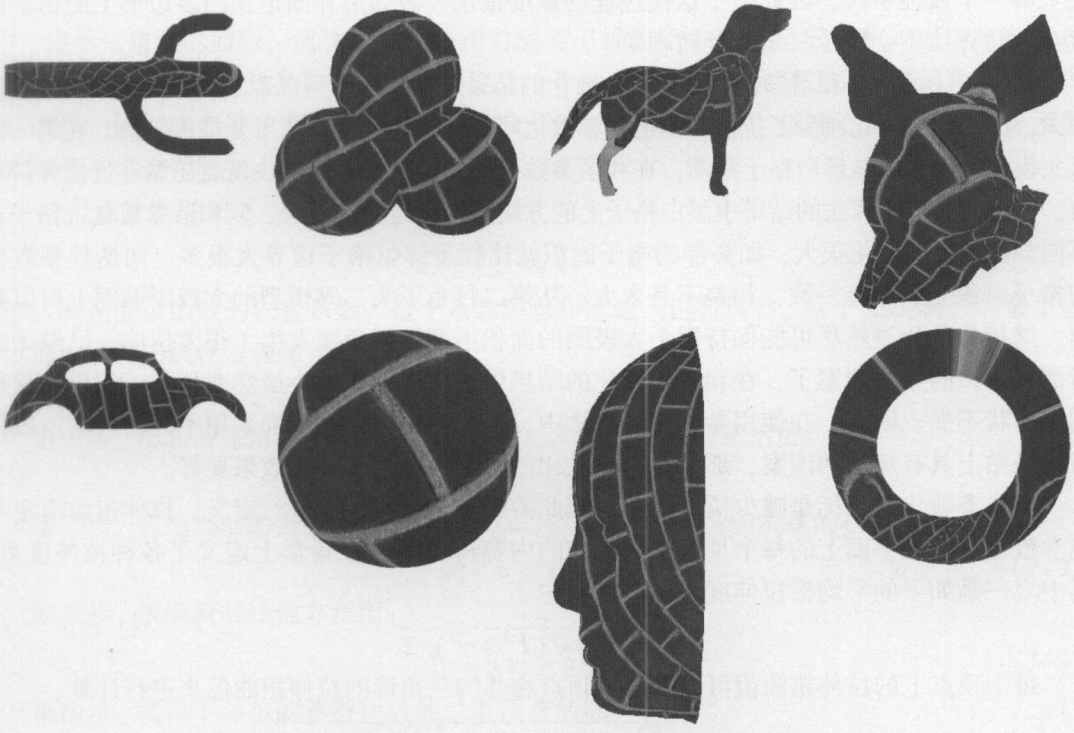


图 6-8　多层虚拟边界参数化贴图

第7章 迭代参数化算法

7.1 算法设计

和谐参数化的设计目标就是尽可能地保持角度不变，没有把拉伸考虑进去。也就是和谐参数化在固定边界的前提下，尽可能地保持角度不变，但这样会产生很大的拉伸。造成贴图的效果不好。为了减少拉伸，一种算法设计的思路是以拉伸度量为目标设计一个优化问题，这个优化问题的优化解就能够尽可能地减少拉伸。但拉伸度量和参数化坐标是非线性关系，从而得到的系统是非线性的。另一种算法设计思路是在和谐参数化得到结果之后，根据三维模型当前的拉伸度量，循环迭代更新结果，从而减少拉伸度量。这是一种循序渐进的方法。因此，迭代参数化的算法为了解决保角参数化结果产生的拉伸问题，首先进行保角参数化，然后在当前参数化的基础上，经过多次迭代把局部的拉伸重新分布到周围顶点。每次迭代只要求解一个线性系统，因此整个迭代过程的速度很快。假如边界固定在凸多边形上的话，迭代参数化算法不会产生三角形翻转现象。

如图7-1所示是和谐参数化和迭代参数化的结果对比，从中可以看出，和谐参数化的拉伸很大，而迭代参数化减少了拉伸。但迭代参数化同时在保持角度上效果变得比较差。在第一行人头模型上采用的是黑白格子贴图，在和谐参数化中黑白格子上的方块角度还能够近似保持垂直，而迭代参数化算法的结果中黑白格子上的方块角度已经变化很大。但和谐参数化上格子在不同部位的面积变化很大，如头部的格子面积就比脖子部位格子边界大很多。而迭代参数化的格子面积能够保持一致，相差不是太大。从第二行兔子头三维模型的太极图贴图上可以看出，迭代参数化虽然尽可能保持每个太极图的面积不变，但角度发生了很大扭曲，已经无法看清楚原来的太极图案了，在和谐参数化的结果中，太极图的大小虽然变化了，但仍能保持图案形状不变。因此，在使用参数化的算法中，需要根据目标的不同采用不同的算法。假如目标是贴上具有规则的图案，那么和谐参数化的结果比迭代参数化效果要好。

迭代参数化的目标是减少拉伸扭曲，因此需要先对拉伸扭曲进行定义。拉伸扭曲值定义在参数化后二维平面上的每个顶点上。之前的内容讲到每个三角形上定义了各种拉伸度量。其中之一是如下的平均根拉伸度量：

$$\sigma(U) = \sqrt{(\Gamma^2 + \gamma^2)/2}$$

每个顶点上的拉伸扭曲值可以用这个顶点相邻的三角形的拉伸扭曲值来进行计算：

$$\sigma_i = \sqrt{\sum A(T_j)\sigma(U_j)^2 / \sum A(T_j)}$$

也就是这个顶点相对应的三维模型上顶点的邻居三角形的拉伸扭曲值求和。其中，

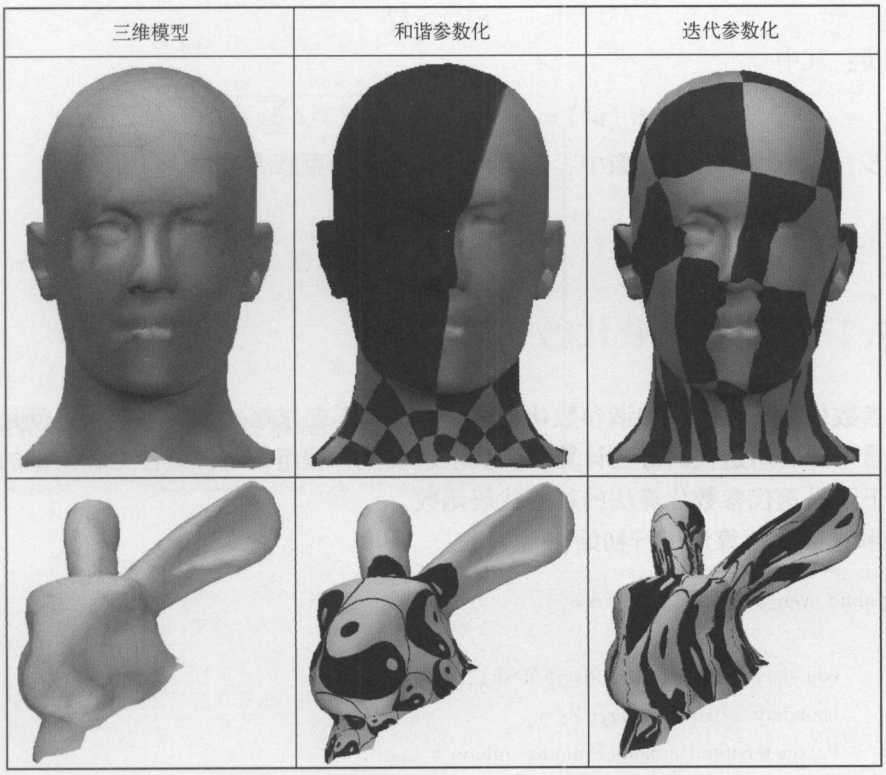


图7-1　和谐参数化和迭代参数化算法贴图

$A(T_j)$表示三角形的面积，但这个面积是相对应的三维模型上的三角形面积，而不是参数化后二维平面的面积。迭代参数化设计通过定义一个能量函数，然后对这个能量函数求导，得到它的最小值。这个最小值就是位置的参数化后的二维顶点坐标。但迭代参数化的能量函数系数随着迭代进行调整。

能量函数定义如下：

$$E(u_i) = \sum_j w_{ij} (u_j - u_i)^2 \tag{7-1}$$

式中，w_{ij}是能量函数的权重系数，这个系数随着每次迭代进行调整。这个二次函数的最小值可以通过求解的线性方程组得到。

系统构建步骤如下。

第一步：用和谐保角参数化计算参数化后二维顶点坐标的初始值：

$$u^0 = \{u_i^0\} \tag{7-2}$$

第二步：通过当前的参数化坐标位置 $u^h = \{u_i^h\}$，计算能量函数新的权重：

$$w_{ij}^{h+1} = w_{ij}^h / (u_j^h) \tag{7-3}$$

第三步：求解新的线性方程组：

$$\sum_j w_{ij}(u_j - u_i) = 0 \tag{7-4}$$

第四步：获得下一步的参数化后的二维顶点坐标：

$$u^{h+1} = \{u_i^{h+1}\} \tag{7-5}$$

第五步：重复第二步到第四步，直到满足如下条件：

$$E_s^{h+1} \geqslant E_s^h \tag{7-6}$$

第六步：其中

$$E_s^h = E_s(u^h) = \sqrt{\sum A(T)\sigma\,(U^h)^2 / \sum A(T)} \tag{7-7}$$

第七步：在上面的权重更新中，也可以选择如下的更新方法：

$$w_{ij}^{new} = w_{ij}^{old} / \sigma_j^{\eta} \tag{7-8}$$

第八步：其中$\{\sigma_i^j\}$，$0 \leqslant \eta \leqslant 1$。

7.2 迭代参数化代码

迭代参数化算法代码和和谐参数化算法类似，都需要求解线性方程组，也就是需要构建矩阵和向量。然后用迭代的方法计算，多次求解线性方程组。每次线性方程组里的矩阵也需要更新。下面是迭代参数化算法的核心代码函数。

1. 用和谐参数化算法进行初始化

```
public override void Parameterize( )
{
    boundary = new ParaBoundary( Mesh) ;
    boundary. SetUpBoundary( ) ;
    ParameterzitionHarmonicUniform uniform =
        new ParameterzitionHarmonicUniform( Mesh) ;
    uniform. Parameterize( ) ;
    uniform. Update( EnumParaShowType. Flat) ;
    RightB = uniform. BuildRightB( boundary) ;
    SparseMatrix leftMatrix =
        uniform. BuildMatrixLeft( EnumParaHarmonicType. Uniform, boundary) ;
    Iterat( leftMatrix) ;
}
```

2. 计算顶点的拉伸

```
public double[ ] ComputeVertexStrech( TriMesh orig, TriMesh flat)
{
    double[ ] rms = ComputeStretchRmsL2( orig, flat) ;
    double[ ] areaTarget = TriMeshUtil. ComputeAreaFace( flat) ;

    double[ ] vetexStretch = new double[ orig. Vertices. Count] ;

    foreach ( TriMesh. Vertex v in orig. Vertices)
    {
        double areaSum = 0;
        double stretchSum = 0;

        foreach ( TriMesh. Face adjF in v. Faces)
        {
```

```
                double area = areaTarget[ adjF. Index ];
                double stretch = rms[ adjF. Index ];
                areaSum += area;
                stretchSum += area * stretch * stretch;
            }
            vetexStretch[ v. Index ] = Math. Sqrt( stretchSum/areaSum );
        }

        return vetexStretch;
    }
```

3. 更新权重，即更新线性系统的矩阵

```
    private void UpdateMatrix( SparseMatrix leftMatrix )
    {
        double[ ] vertexStretch =
        ParameterizationMeasure. Instance. ComputeVertexStrech( backupMesh, Mesh );
            foreach ( TriMesh. Vertex vi in this. Mesh. Vertices)
            {
                if ( vi. OnBoundary)
                {
                    continue;
                }
                leftMatrix[ vi. Index, vi. Index ] = 0;
                foreach ( TriMesh. Vertex vj in vi. Vertices)
                {
                    double weight = leftMatrix[ vi. Index, vj. Index ]
                                / vertexStretch[ vj. Index ];
                    leftMatrix[ vi. Index, vj. Index ] = weight;
                    leftMatrix[ vi. Index, vi. Index ] -= weight;
                }
            }
    }
```

4. 用新的矩阵进行迭代，更新参数化结果

```
    public void Iterat( SparseMatrix leftMatrix)
    {
        for ( int i = 0; i < ConfigParam. Instance. IterativeCount; i ++ )
        {
            UpdateMatrix( leftMatrix) ;
            SolveLinearSystem( leftMatrix, RightB) ;
            UpdatePosition( ) ;
        }
    }
```

7.3 迭代参数化效果分析

迭代参数化能够减少拉伸，但同时也就增大了角度扭曲。如图 7-2 所示，从初始化后的贴图上可以看出，本来大小一样的黑白格子变得大小不一致，但格子边长还是保持近似直

序号	迭化参数化	贴图
初始化		
1次迭代		
2次迭代		
5次迭代		
10次迭代		

图 7-2　兔子头模型迭代参数化过程

线，同时夹角还是近似 90°。而经过迭代后，格子的大小变得越来越一致，但边长无法继续保持直线，同时角度扭曲也越来越大。

由于迭代算法不能保持角度不变，因此在贴图应用中某些情况下的效果不好。如图 7-3 所示，贴图是八卦圆形，由于迭代算法造成的角度扭曲，原来的八卦图案随着迭代次数的增加，越来越扭曲。但对于其图案，如花卉，由于迭代算法减少了拉伸，因此效果较好。

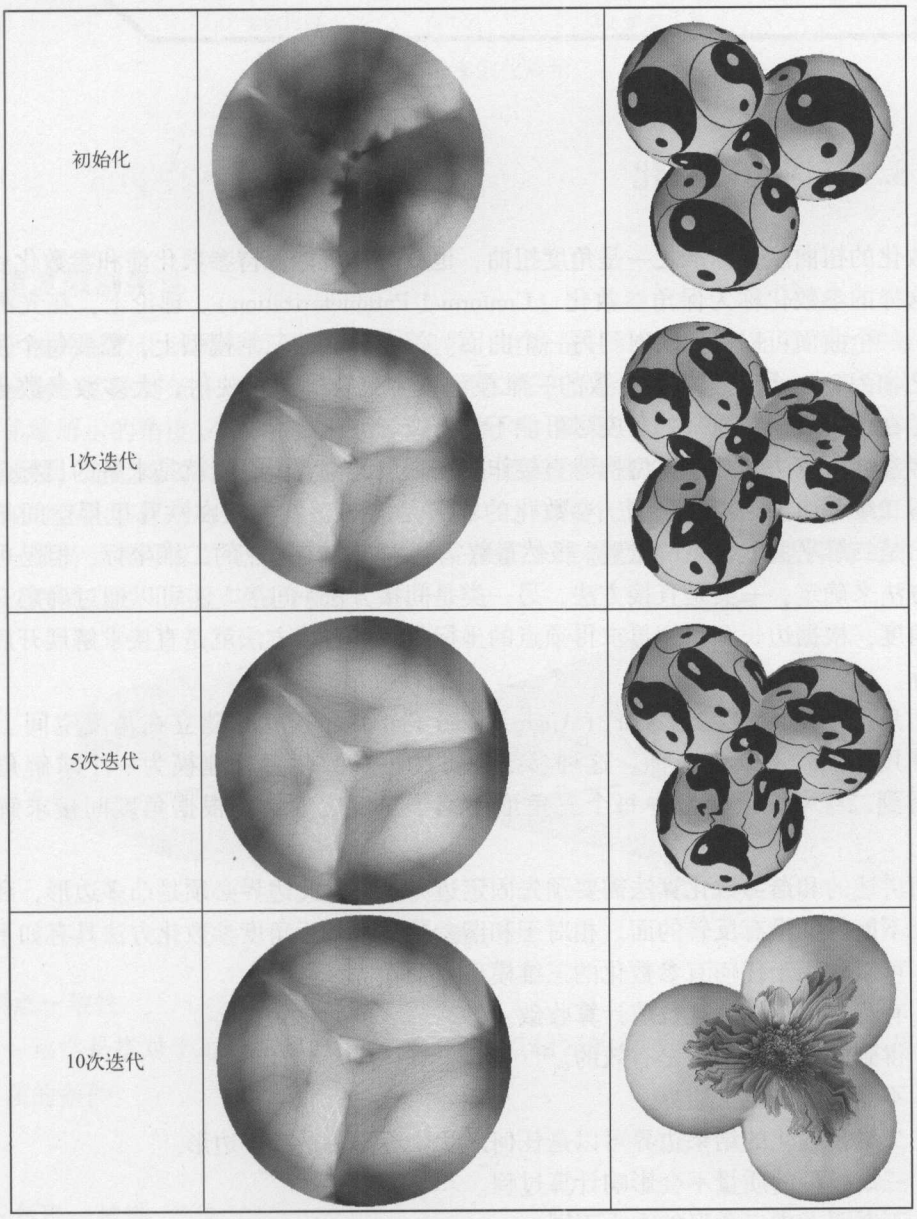


图 7-3　3 个半球模型的迭代参数化过程

第8章 基于角度平展参数化算法

8.1 保角参数化

参数化的扭曲度量标准之一是角度扭曲，也就是尽可能保持参数化前和参数化后的角度不变。这样的参数化称为保角参数化（Conformal Parameterization）。理论上，在光滑曲面的情况下，一个曲面可以保角映射到另一个曲面。但在离散的三维模型上，需要每个顶点相邻的角度之和为2π，因此只有极少数的三维模型具有完全的保角映射。大多数参数化算法的目标就是在映射后保持原来的角度尽可能不变，或者变化尽可能小。

很多参数化算法的数学模型构建直接作用在顶点的位置上，也就是求解的目标是三维模型展开后二维平面上顶点的位置。参数化的输入是已知三维模型在欧几里得空间的顶点坐标，输出是二维平面上顶点的位置。虽然最终需要确定每个顶点的二维坐标，但是可以分为两大类方法来确定，一类是直接方法，另一类是间接方法。间接方法可以通过确定三角形的边长或角度，根据边长和角度再求得顶点的坐标位置。直接方法就是直接求解展开后顶点的位置坐标。

基于角度平展的参数化 ABF（Angle Based Flattering）就是建立在角度空间上，而不是直接作用于三维的顶点坐标。这种参数化方法把参数化过程建模为一个求解角度的模型，在得到二维平面上模型中每个三角形的三个角度之后，再根据角度间接求解顶点的位置。

前面讲述的和谐参数化算法需要预先固定边界，并且此边界必须是凸多边形，和谐参数化的结果不能保证没有反转的面。相对于和谐参数化，基于角度参数化方法具有如下优点。

（1）可以作用于任何有参数化的三维模型。

（2）有优化解存在并且数值计算收敛。

（3）得到的参数化结果是有效的。

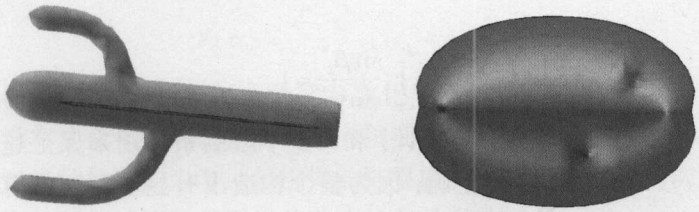
（4）不需要预先定义边界。

（5）二维参数化的结果边界可以是任何形状，不需要是凸多边形。

（6）三维模型的质量不会影响计算过程。

（7）不需要考虑三角形的大小不同。

ABF 参数化效果如图 8-1 所示。从中可以看出，参数化后的边界是不固定的形状，仙人掌的三维模型切开后再映射到平面上，相当于沿着切口把仙人掌展开。



（a）参数化前　　（b）参数化后

图 8-1　ABF 参数化展示

8.2　角度空间数学系统构建

8.2.1　角度限制条件

角度平展的参数化系统是从局部开始构建的，也就是对于每个顶点相邻的边和每个三角形的角度设定一些限制条件。在三维模型展开到二维平面后，对于如图 8-2 所示的每个顶点的一层邻域所示的角度，需要满足 3 个的限制条件：**顶点一致性、三角形一致性、顶点邻居轮型一致性**。通过这 3 个限制条件可以保证每个顶点和周围的相邻三角形的夹角尽可能保持不变，同时还能够保持相邻的三角形结构不变。这样每个顶点局部都能够保持角度不变，那么整个三维模型就可以尽可能保持角度不变。虽然三维模型展平后，无法保持每个角度不变，但可以把误差尽可能地均匀分配。

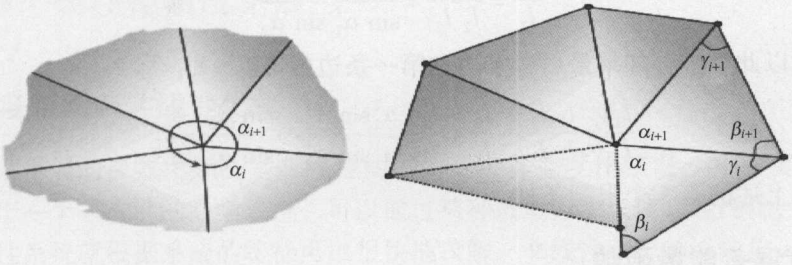


图 8-2　顶点邻域

1. 顶点一致性

顶点一致性是指对于每个内部的顶点 v 和此顶点相邻的角度 α_1^*，…，α_d^*，这些角度需要满足下面的条件：

$$\sum_{i=1}^{d} \alpha_i^* = 2\pi$$

2. 三角形一致性

三角形一致性是指对于每个三角形面的 3 个角 α^*、β^*、γ^*，需要满足：

$$\alpha^* + \beta^* + \gamma^* = \pi$$

3. 顶点邻居轮型一致性

对于每个内部顶点，左边的角度 $\beta_1^*, \cdots, \beta_d^*$ 和右边的角度 $\gamma_1^*, \cdots, \gamma_d^*$，需要满足如下

条件：

$$\prod_{i=1}^{d}\frac{\sin\beta_i^*}{\sin\gamma_i^*}=1$$

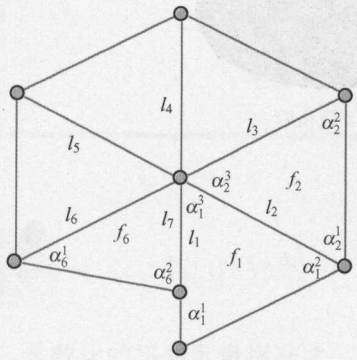


图 8-3　第 3 个限制条件推导图

第 1 个和第 3 个限制条件用来保证这个模型的拓扑结构正确，因为整个构造没有显示对拓扑有约束，这两个条件隐式的约束拓扑结构。满足第 1 个条件确保每个顶点和它的邻居顶点在一个平面上，也就是从每个顶点的第一条边开始逆时针旋转到最后一条边正好和第一条边重合。满足条件 3 确保这个顶点的一层邻域是封闭的，也就是每个顶点的第一条边和最后一条边长度一样，如图 8-3 所示。

第 3 个限制条件的推导过程如下。

第一步：在图 8-3 中，假如确定了 l_1 的边长，在面 f_1 中已知两个角度 α_1^1 和 α_1^2，那么有：

$$\frac{l_1}{l_2}=\frac{\sin\alpha_1^2}{\sin\alpha_1^1} \tag{8-1}$$

第二步：在面 f_2 中，可以得到：

$$\frac{l_2}{l_3}=\frac{\sin\alpha_2^2}{\sin\alpha_2^1} \tag{8-2}$$

第三步：综合上述两个方程，可以得到：

$$\frac{l_1}{l_3}=\frac{l_1}{l_2}\frac{l_2}{l_3}=\frac{\sin\alpha_1^2}{\sin\alpha_1^1}\frac{\sin\alpha_2^2}{\sin\alpha_2^1} \tag{8-3}$$

第四步：以此类推，如果最后一条边和第一条边相等 $l_7=l_1$，那么：

$$\frac{l_1}{l_7}=\frac{l_1}{l_2}\frac{l_2}{l_3}\cdots\frac{l_6}{l_7}=\frac{\sin\alpha_1^2\sin\alpha_2^2}{\sin\alpha_1^1\sin\alpha_2^1}\cdots\frac{\sin\alpha_6^2}{\sin\alpha_6^1}=1 \tag{8-4}$$

从而得到上述的第 3 个限制条件。

8.2.2　角度误差能量函数

每种参数化算法都需要设计一个能量函数，这个函数度量扭曲的程度。扭曲的值通常是定义在局部的顶点、面或者边上。这些局部的扭曲值之和就构成了一个全局的扭曲值。这个能量函数的变量是三维模型参数化映射后在二维平面上的坐标，通过求解使此能量函数最小化来得到一个优化的解。也就是得到的三维模型所有顶点的二维坐标在所有可能的解中，能够使特定的能量函数达到最小值。根据能量函数的不同，得到的结果不一样。

基于角度的参数化（ABF）算法的思路是把参数化过程构建为一个有约束的最小化模型。参数化需要减少角度和边界的扭曲，因此需要构造可靠稳定和自由边界。基于角度平展的参数化 ABF（Angle Based Flattering）把参数化问题构建为一个直接作用在角度上的非线性的优化问题。

首先定义一个目标函数，这个目标函数度量三维模型上每个角的角度在参数化前和参数化之后的变化，用角度之差的平方来计算。

$$F(\alpha)=\sum_{i=1}^{P}\sum_{j=1}^{3}(\alpha_i^j-\phi_i^j)^2w_i^j \tag{8-5}$$

式中，α、ϕ 分别表示三维模型上每个角度参数化后和参数化之前的角度值；w 是相应角度上的权重。假如没有约束条件的话，展开后的角度值和原来角度一样才能满足上述函数的最小值，也就是上述函数最小值为零。但这样的结果破坏了三维模型的结构，因此需要加上约束条件。从而得到如下的基于角度平展参数化的能量函数：

$$F(\alpha)+\sum_{i=1}^{P}\lambda_i g_i^{(2)}(\alpha)+\sum_{k=1}^{M_{\text{int}}}\mu_k g_k^{(3)}(\alpha)+\sum_{k=1}^{M_{\text{int}}}v_k g_k^{(4)}(\alpha) \tag{8-6}$$

式中，λ_i、μ_k、v_k 分别表示权重系数；M_{int}是内部顶点的数量。

在 ABF 参数化模型构造中，未知数为所有三角形参数化后在平面上的角度，第 1、2 个条件对于角度是线性的约束，第 3 个条件是非线性的约束，所以整个问题模型对于角度是非线性的。可以用牛顿迭代法求解上述能量函数的最优值：

$$\begin{aligned}&\|\nabla F(x)\|>\varepsilon\\&\nabla^2F(x)\delta=-\nabla F(x)\\&x\leftarrow x+\delta\end{aligned} \tag{8-7}$$

海斯（Hessian）矩阵$\nabla^2F(x)$大小是 $4nf+2n_{\text{int}}$，nf 是面数，未知数变量个数是 $3nf$，$\nabla^2F(x)\delta=-\nabla F(x)$是一个线性系统。

ABF 算法得到的结果可能会出现边界上相交的情况。可以通过加上新的约束条件到迭代过程中来解决这个问题。

8.3　线性化角度平展

8.3.1　角度限制条件线性化

基于角度平展的参数化算法是非线性的，因此计算速度比较慢，并且实现起来比较复杂。通常对于一个非线性的数学系统，可以通过某种近似方法来使它线性化，得到一个线性的系统。线性系统就根据系统的特征可以很快地求解。非线性系统的线性化一般来说有两种思路：一种思路是把非线性系统得到的公式进行线性化；另一种思路是修改原来的模型，使系统本身就成为一个线性的系统。为了把 ABF 进行线性简化，很多改进的算法通常是把 ABF 构造模型的数学公式进行线性简化，但这种简化的方式会使方程的解准确度受到影响。

在 ABF 算法中，未知数是每个角的角度。如果把未知数变为角度的误差，然后把 ABF 重新用误差角度来进行构建，就可以得到一个线性的模型。这种把原来的非线性模型简化为线性模型的误差是角度误差的二次形式。这样得到的线性系统的误差是可控制的。这种线性角度平展的（Linear ABF）参数化也称误差调整的方法（Error Adjust）。这种线性化的方法在构建模型时就是线性的，因此相对于先构建非线性的模型，在数值计算时每一步对非线性的计算做线性近似求解来说，有模型简单、计算可靠、误差可控制的优越性。如果某个顶点的邻居三角形中有钝角三角形，则线性 ABF 生成无效（Invalid）的解，但可以用多次迭代的方法，每次把角度限制在 0 ~ π 之间，来生成有效的解。

如果用 α^* 表示 ABF 的最优解，α 表示初始解，那么最优解和初始解之间有个误差：

$$\alpha^* = \alpha + e_\alpha \tag{8-8}$$

从而7.2节中的3个一致性约束条件可以重构如下。

1. 顶点一致性

对于内部顶点 v 和相邻的角度 $\alpha_1,\cdots,\alpha_d$，有：

$$\sum_{i=1}^{d} e_i = 2\pi - \sum_{i=1}^{d} \alpha_i \tag{8-9}$$

2. 三角形一致性

对于一个三角形的三个角度 α、β、γ，有：

$$e_\alpha + e_\beta + e_\gamma = \pi - (\alpha + \beta + \gamma) \tag{8-10}$$

3. 顶点邻居轮型一致性

对于内部顶点 v 的左边角度 $\beta_1,\cdots,\beta_d$ 和右边角度 $\gamma_1,\cdots,\gamma_d$，有：

$$\sum_{i=1}^{d} \lg(\sin\beta_i + e_{\beta i}) - \lg(\sin\gamma_i + e_{\gamma i}) \tag{8-11}$$

顶点轮型一致性约束条件的线性化如下所述。

上述的重构中，初始角度是已知变量，未知变量是角度误差，但和ABF一样，对于角度误差变量仍是非线性的，并且对于第3个条件来说，是更复杂的非线性关系。因此需要对这个约束条件进行线性化。

第一步：根据泰勒展开公式

$$\lg[\sin(\alpha + e)] = \lg(\sin\alpha) + \cot(a)e - \frac{1}{2}[1 + \cot^2\alpha]e^2 + \cdots \tag{8-12}$$

第二步：上述公式可以近似为

$$\lg[\sin(\alpha + e)] \approx \lg(\sin\alpha) + \cot(\alpha)e \tag{8-13}$$

第三步：这个近似导致的误差是角度误差变量的二次形式。也就是说，对于一个很小的角度误差 e，这个约束造成的误差更小，也就是10^{-3}量级上的角度误差，只会造成早约束条件上10^{-6}量级的误差。这点是线性化成功的关键。

第四步：从而第3个约束条件从非线性变为线性的：

$$\sum_{i=1}^{d} \cot(\beta_i)e_{\beta_i} - \cot(\gamma_i)e_{\gamma_i} = \sum_{i=1}^{d} \lg(\sin\gamma_i) - \lg(\sin\beta_i) \tag{8-14}$$

在这个线性模型中，第1个条件等式右边度量初始值对于平面约束的误差，第2个条件度量对于三角形约束的误差，第3个条件度量轮型约束的误差。从而给定一个初始的参数化角度，就可以得到一个对于这个初始角度和最优角度的误差变量的线性模型。初始的参数化角度可以使用三维模型参数化之前的角度。

8.3.2 线性ABF系统构建

在约束条件从非线性经过泰勒展开变为线性之后，就可以构建一个线性系统了。这个线性系统的构建也是需要构建线性系统左边的矩阵和右边的向量。线性系统的未知数是角度误差。

第一步：对于ABF参数化目标优化函数

$$F(\alpha) = \sum_{i=1}^{N} \frac{1}{\alpha_i^2}(\alpha_i^* - \alpha_i)^2 \tag{8-15}$$

第二步：可以重构如下优化函数

$$\text{minimize} \sum_{i=1}^{N} \frac{1}{\alpha_i^2} e_i^2, \quad Ae = b \tag{8-16}$$

第三步：假如引入一个新的变量

$$r_i = \frac{e_i}{\alpha_i} \tag{8-17}$$

第四步：上述变量转换的矩阵形式为

$$e = \boldsymbol{D}_a r \tag{8-18}$$

第五步：其中 $\boldsymbol{D}_a = \text{diag}(\alpha_i)$ 是一个对角矩阵。对角线上的角度值。从而优化函数转换为

$$\text{minimize} \|r\|^2 \quad \text{subject to } \boldsymbol{C}_r = \boldsymbol{b} \tag{8-19}$$

第六步：其中 $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{D}_a$ 是一个 $(n_t + 2n_i) \times 3n_t$ 的矩阵。n_t 是三角形数量，n_i 是内部顶点数量。因为上个约束条件是互相独立的，因此矩阵 $\boldsymbol{C}$ 是满秩（Full Rank）的。矩阵的行数小于矩阵的列数，这是一个最小范数问题（Least - Norm Problem）。

第七步：因此上述的最小范数问题有唯一的最优解为：

$$r = \boldsymbol{C}^{\mathrm{T}} (\boldsymbol{C}\boldsymbol{C}^{\mathrm{T}})^{-1} \boldsymbol{b} \tag{8-20}$$

第八步：求解线性系统过程。

（1）先求解如下的以中间变量 x 为未知数的线性系统：

$$(\boldsymbol{C}\boldsymbol{C}^{\mathrm{T}})\boldsymbol{x} = \boldsymbol{b} \tag{8-21}$$

（2）然后根据 $\boldsymbol{x}$ 得到 $\boldsymbol{r}$：

$$\boldsymbol{r} = \boldsymbol{C}^{\mathrm{T}}\boldsymbol{x} \tag{8-22}$$

（3）再根据 $\boldsymbol{r}$ 得到误差 e。

（4）最后根据 e 和初始角度得到最优角度。

1. 初始角度

初始角度的选择可以选择为三角形在三维空间没有参数化之前的角度，这样有时候会造成无效的角度解。例如对于一个顶点的角度远远小于 2π 的情况，就会出现无效的角度解。可以顶一个门槛值，例如 1，假如一个顶点相邻的角度和小于 1。由于离散情况下三维模型内部顶点的相邻角度之和不等于 2π，因此需要把内部顶点相邻的角度进行缩放，从而使这些角度之和为 2π。对于边界顶点来说，可以采用原来的值作为初始值。如果用 α_i^0 来表示原来的三维角度，初始角度设置如下：

$$\alpha_i = \begin{cases} \alpha_i^0 \dfrac{2\pi}{\sum_{i=1}^{d} \alpha_i^0} & \text{内部顶点} \\ \alpha_i^0 & \text{边界顶点} \end{cases} \tag{8-23}$$

2. 线性 ABF 算法流程

（1）计算初始角度 α。

（2）根据 3 个约束条件，得到线性方程组：

$$Ae_\alpha = b \tag{8-24}$$

（3）计算：

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{D}_{\alpha} \tag{8-25}$$

（4）求解：

$$(\boldsymbol{C}\boldsymbol{C}^{\mathrm{T}})\boldsymbol{x} = \boldsymbol{b} \tag{8-26}$$

（5）计算角度误差：

$$e_a = \boldsymbol{D}_a \boldsymbol{C}^{\mathrm{T}} \boldsymbol{x} \tag{8-27}$$

（6）得到最终的优化角度：

$$\alpha^* = \alpha + e_a \tag{8-28}$$

8.4 角度平展参数化代码

1. 初始化

由于3个约束条件中顶点一致性和顶点邻居轮型一致性条件只作用在内部顶点上，因此需要构建数据结构来确认内部节点。这步是线性化ABF算法复杂的地方。因为线性系统的顶点和三维模型的顶点不一致，需要区分内部顶点和外部顶点，而三维模型顶点的排序没有区分内部顶点和外部顶点，因此打乱了原来三维模型顶点的排序。在初始化时需要构建两个数组来保存从三维模型顶点抽取内部顶点之后的映射。也就是在求得线性系统的内部顶点的解之后，顶点的序号还需要根据事前保存的数组映射到三维模型顶点的序号上。

1）半边映射

```
private int[ ] hfOld2New;
private int[ ] vOld2New;
private int interVertice;

private int[ ] ReMapHalfEdge( )
{
    int[ ] hfOld2NewIndex = new int[ Mesh. HalfEdges. Count ];
    int count = 0;
    foreach ( TriMesh. HalfEdge item in Mesh. HalfEdges)
    {
        if ( item. OnBoundary)
        {
            hfOld2NewIndex[ item. Index ] = -1;
        }
        else
        {
            hfOld2NewIndex[ item. Index ] = count;
            count ++ ;
        }
    }
    return hfOld2NewIndex;
}
```

2）顶点映射

```
private int[ ] ReMapVertice( )
{
    int[ ] vOld2NewIndex = new int[ Mesh. Vertices. Count ];
    int count = 0;
    foreach ( TriMesh. Vertex item in Mesh. Vertices)
    {
        if ( item. OnBoundary)
        {
            vOld2NewIndex[ item. Index ] = -1;
        }
        else
        {
            vOld2NewIndex[ item. Index ] = count;
            count ++ ;
        }
    }
    return vOld2NewIndex;
}
```

3）计算内部顶点数量

```
private int ComputeInterVertice( )
{
    int count = 0;
    foreach ( TriMesh. Vertex item in Mesh. Vertices)
    {
        if ( !item. OnBoundary)
        {
            count ++ ;
        }
    }
    return count;
}
```

4）初始化

```
private void Init( )
{
    hfOld2New = ReMapHalfEdge( ) ;
    vOld2New = ReMapVertice( ) ;
    interVertice = ComputeInterVertice( ) ;
}
```

2. 初始角度

对于f个面的三维模型来说，具有$3f$个角。下面函数计算初始的三维模型角度。

```
public double[ ] ComputeInitAngles( )
{
    int n = 3 * Mesh. Faces. Count;
    double[ ] initAngle = new double[ n ];
    for ( int i = 0; i < n; i ++ )
    {
        initAngle[ i ] = 0;
    }

    foreach ( TriMesh. Vertex v in Mesh. Vertices)
    {
        double sum = 0;
        foreach ( TriMesh. HalfEdge item in v. HalfEdges)
        {
            if ( item. OnBoundary)
            {
                continue;
            }
            TriMesh. HalfEdge nextItem = item. Previous. Opposite;
            double angle = TriMeshUtil. ComputeAngle( item, nextItem) ;
            sum += angle;
        }
        foreach ( TriMesh. HalfEdge item in v. HalfEdges)
        {
            if ( item. OnBoundary)
            {
                continue;
            }
            TriMesh. HalfEdge nextItem = item. Previous. Opposite;
            double angle = TriMeshUtil. ComputeAngle( item, nextItem) ;
            if ( item. FromVertex. OnBoundary)
            {
                initAngle[ hfOld2New[ item. Index] ] = angle;
            }
            else
            {
                initAngle[ hfOld2New[ item. Index] ] = angle * 2 * Math. PI/sum;
            }
        }
    }
    return initAngle;
}
```

计算对角矩阵。

```
public SparseMatrix BuildMatrixDa(double[ ] initAngle)
{
    SparseMatrix Da = new SparseMatrix(initAngle. Length,initAngle. Length);
    for (int i =0;i < initAngle. Length;i ++)
    {
        Da[i,i] = initAngle[i];
    }
    return Da;
}
```

3. 限制条件

线性化的 ABF 首先需要通过 3 个限制条件构建线性系统的左边矩阵和右边的向量。如果三维模型有 f 个面、m 个内部顶点，那么这个线性系统矩阵大小是 $3f \times (2m + f)$。下面函根据线性化的 3 个限制条件构建矩阵和向量。

（1）顶点一致性的限制条件，也就是对于每个内部顶点都要保持相邻的角度之和是 2π。

```
foreach (TriMesh. Vertex v in Mesh. Vertices)
{
    if (v. OnBoundary)
    {
        continue;
    }
    double sum =0;
    foreach (TriMesh. HalfEdge hf in v. HalfEdges)
    {
        if (hf. OnBoundary)
        {
            continue;
        }
        A[vOld2New[v. Index],hfOld2New[hf. Index]] =1;
        sum += initAngle[hfOld2New[hf. Index]];
    }
    RightB[v. Index] =2 * Math. PI - sum;
}
```

（2）三角形一致性的限制条件，也就是三角形的内角和是 π。

```
int baseIndex = interVertice;
foreach (TriMesh. Face face in Mesh. Faces)
{
    double sum =0;
    foreach (TriMesh. HalfEdge hf in face. Halfedges)
    {
```

```
            double angle = initAngle[hfOld2New[hf.Index]];
            sum += angle;
            A[baseIndex + face.Index, hfOld2New[hf.Index]] = 1;
        }
        RightB[baseIndex + face.Index] = Math.PI - sum;
    }
```

（3）顶点轮型一致性限制条件，也就是每个顶点周围的邻居顶点围绕此顶点绕一圈后，能够回到原来的顶点。

```
baseIndex += Mesh.Faces.Count;
foreach (TriMesh.Vertex v in Mesh.Vertices)
{
    if (v.OnBoundary == true)
    {
        continue;
    }
    double sum = 0;
    foreach (TriMesh.HalfEdge hf in v.HalfEdges)
    {
        if (hf.OnBoundary)
        {
            continue;
        }
        TriMesh.HalfEdge hfNext = hf.Next;
        TriMesh.HalfEdge hfNextN = hfNext.Next;

        double betaAngle = initAngle[hfOld2New[hfNext.Index]];
        double thetaAngle = initAngle[hfOld2New[hfNextN.Index]];

        int row = baseIndex + vOld2New[v.Index];
        int col1 = hfOld2New[hfNext.Index];
        int col2 = hfOld2New[hfNextN.Index];

        A[row, col1] = 1/Math.Tan(betaAngle);
        A[row, col2] = -1/Math.Tan(thetaAngle);
        sum += (Math.Log(Math.Sin(thetaAngle))
                - Math.Log(Math.Sin(betaAngle)));
    }
    RightB[baseIndex + vOld2New[v.Index]] = sum;
}
```

4. 求解线性系统

在得到上述的初始角度、系统矩阵之后，可以构建线性系统，并求解。在得到角度之后，再通过重构的方式得到最终顶点的位置。

```
public override void Parameterize( )
{
    Init( );
    double[ ] initAngle = ComputeInitAngles( );
    double[ ] RightB = null;
    SparseMatrix A = BuildLeftMatrixAndRightB( ref initAngle, out RightB);
    SparseMatrix Da = BuildMatrixDa( initAngle);
    SparseMatrix C = HackMultiply( A, Da);
    double[ ] r = LinearSystem. Instance. SloveLeastNormal( ref C, ref RightB);
    double[ ] newAngle = ComputeFinalAngle( Da, r, initAngle);
    Reconstruction( newAngle);
}

private double[ ] ComputeFinalAngle( SparseMatrix Da,
                                     double[ ] r,
                                     double[ ] initAngle)
{
    double[ ] errorAngle = Da. Multiply( r);
    double[ ] newAngle = new double[ 3 * Mesh. Faces. Count];
    for ( int i = 0; i < newAngle. Length; i ++ )
    {
        errorAngle[ i] % = Math. PI;
        newAngle[ i] = initAngle[ i] + errorAngle[ i];
    }
    return newAngle;
}
```

8.5　线性 ABF 效果分析

线性基于角度平展的参数化算法是自由边界的参数化算法，不需要确定边界的位置。因此相对于和谐参数化来说，扭曲程度更小。能够更好地保持原来的角度。线性 ABF 可以作用于各种具有边界大的三维模型上，只要三维模型弯曲不是太大，都可以得到很好的展开结果。

如图 8-4 所示，第 1 行表示三维模型，第 2 行表示和谐参数化结果，第 3 行表示线性 ABF 参数化结果，第 4 行表示线性 ABF 参数化贴图。从中可以看出，这 3 种形状的三维模型在线性参数化后的扭曲比和谐参数化要小，尤其是边界上，因为线性 ABF 参数化的边界是自由边界，因此扭曲就更小。

线性角度平展参数化也可以应用于内部有洞的三维模型，如图 8-5 所示。该图是个三维汽车模型，这个模型具有多个洞，也就是多个边界，采用线性参数化可以成功地把这个三维模型展开到平面，并且多个洞的分布也很均匀。贴图之后的效果比和谐参数化更好。

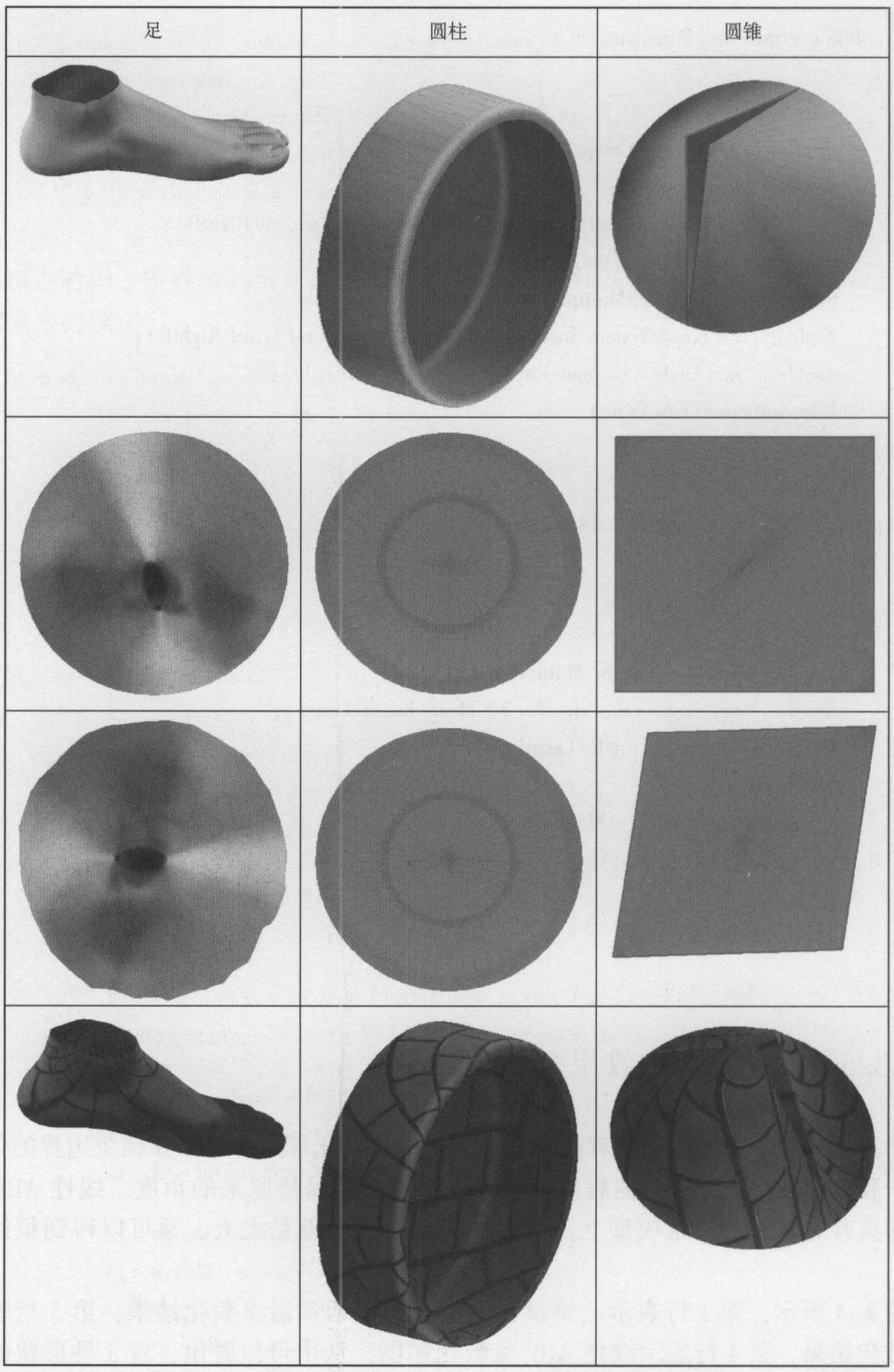
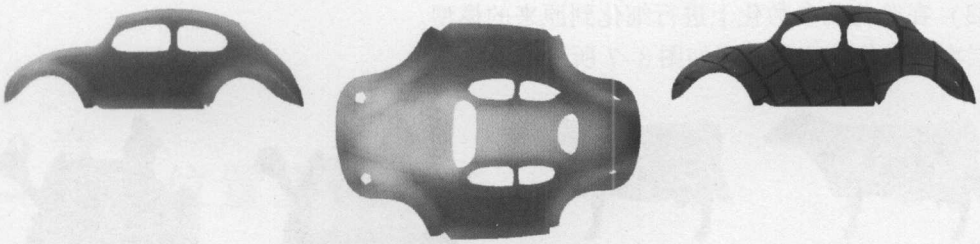


图 8-4　各种形状三维模型线性 ABF 参数化结果

对于弯曲比较大、和平面相差太远的三维模型，线性 ABF 会出现面翻转的情况，如图 8-6 所示，橘红色是背面的颜色。三维模型在展开后，可以看到出现橘红色的面，这表明此面的背面翻转到了前面，和周围的面方向不一致。还会出现相交的情况，如图 8-6 中的第 2 行。因此线性 ABF 算法不适合弯曲的比较大的三维模型。



（a）三维模型　　（b）线性ABF参数化　　（c）贴图

图 8-5　多个边界三维模型线性 ABF 参数化结果

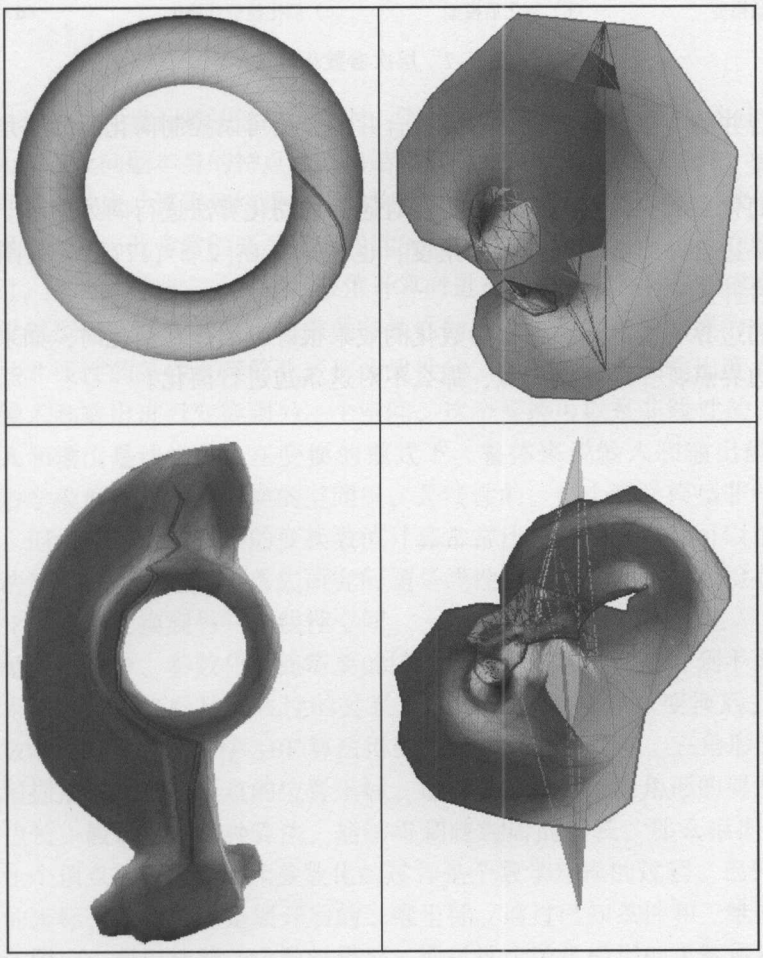


图 8-6　线性 ABF 参数化面翻转图示

层次参数化算法如下所述。

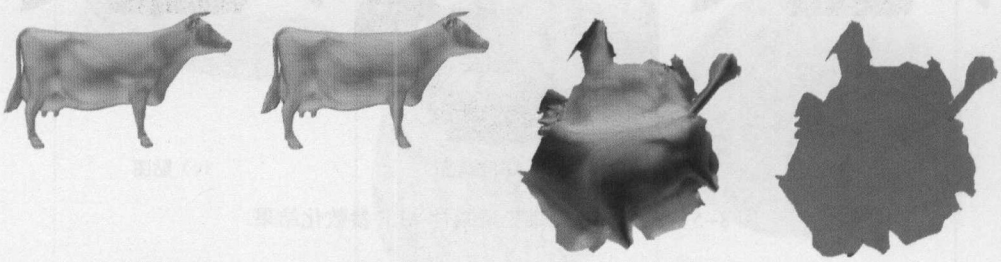
对于顶点比较多的三维网格模型来说，即使是线性的算法，求解线性系统也需要很多时间和内存，因此需要用层次参数化算法（Hierarchical Parameterization）来处理。层次参数化算法分为 3 步。

（1）简化原来的模型。

（2）对简化的模型进行参数化。

（3）在简化的参数化上进行细化到原来的模型。

层次参数化算法的过程如图 8-7 所示。



（a）原始模型　（b）简化后模型　（c）简化模型参数化　（d）细化

图 8-7　层次参数化算法

可以用边合并的方法进行简化，因为边合并的方法可以控制简化后的三角形形状和对简化过程进行记录。

为了控制简化后的三角形的形状，需要对通常的简化算法进行两处修改。

第一：如果边合并后会生成极端的角度，也就是不在[2.5°,177.5°]范围的角度，那么这条边不进行合并。

第二：假如边界不变的话，那么参数化的效果很好，因此在简化时，如果一条边上的两个点一个属于边界点，一个是内部点，那么不对这条边进行简化。

第9章 位置重建算法

9.1 参数化线性系统

在计算机科学中，在设计算法解决问题时，需要根据所面对的问题，构建一个数学模型。根据问题的输入，以及问题本身的特点和输出结果的属性，或者是限制条件，构建一个数学模型。数学模型再经过数值计算得到结果。假如构建的数学模型是非线性的，数值计算就比较复杂，时间慢、误差大。因此常常需要再把非线性的数学模型近似为线性的。在把数学模型从非线性变为线性时，有两种思路：一种是在数值计算时进行化简；另一种是在建模时进行化简。

在构建数学模型时，输入和输出通常是非线性的关系。假如需要构建为线性的关系：一种方法是把某些非线性的条件进行简化近似为线性的，从而使输入和输出变为线性关系；另一种方法是把输入和输出进行变换到另一个空间，这个变换可以是非线性的，但在经过变换的空间里，输入和输出是线性的。在变换的模式下，整体来说输入和输出虽然仍是非线性的，但核心的数学模型位于经过变换的空间里，是线性的。整个数学模型非线性的部分只位于变换的模块，但由于这个非线性的变换数值计算常常比较容易，所以可以忽略。因此，在确定了要用算法解决的问题之后，首先研究问题是线性还是非线性，然后检查能否把非线性的条件进行简化，或者变换到另一个线性空间。

在三维模型参数化中，参数化最终需要的结果是得到三维模型展开到平面的坐标。输入的三维坐标和未知的平面坐标是非线性的关系。线性 ABF 算法经过变换后，在角度空间构建一个线性的数学模型。通过线性 ABF 算法得到的是映射后的角度，三角形 3 个角的角度确定后，就可以根据角度计算出顶点的位置坐标。根据这个思路，三角形的属性除了角度、顶点位置，还有边长。假如应用某种算法，能够得到映射后的边长，那么根据三角形的边长，也可以计算出 3 个顶点的位置。因此参数化的过程是个数学建模的过程，已知条件是三维模型在三维空间的坐标，未知结果是展开后的二维坐标。通过已知条件和三维模型的属性，以及对未知结果的限定，可以构建一个数学模型，通过这个数学模型的求解得到位置的坐标。

常用的思路是直接未知坐标构建一个数学模型，但直接构建常常得到一个非线性的数学模型。因此经过变换后，可以构建一个线性的数学模型。

在线性 ABF 算法中，得到了参数化后的每个三角形的角度，但还需要通过角度计算三角形每个顶点的坐标。角度确定之后，三维网格模型在二维平面上的映射只有旋转、位移、放大 3 个自由度，也就是线性变化自由度。对于一个三维模型来说，经过旋转、位移、缩放后，角度还是保持不变。因此，固定模型上一条边的位置和长度后，整个模型的映射就确定了。

经过角度求解顶点位置有两种算法：一种是贪婪重建算法；另一种是最小二乘重建算法。

9.2 贪婪重建算法

贪婪重建算法（Greedy Reconstruction）首先手工设置第1个三角形的3个顶点在二维平面的位置，然后从第1个三角形开始，根据角度遍历计算所有其他三角形的顶点坐标。第1个顶点位置可以放置在原点，在确定了第1个边的边长之后，根据已知的3个角度，其他2个顶点的位置坐标就可以计算得到。完成当前三角形顶点的位置计算之后，可以接着计算相邻的三角形位置。可以采用堆栈来保存待处理的边，每次从堆栈中得到要进行计算的相邻三角形。在贪婪重建算法中，一个顶点邻区的最后一个三角形不需要计算，因为此时这个三角形的两条边已经在其他三角形中计算过了。贪婪重建算法的缺点是虽然每个顶点的误差很小，但计算每个点的误差会累积，因此后面计算点的误差会很大。在实际实验中，几千个点的模型就会有很明显的误差，3万个点的模型，误差会造成参数化的结果失败。

下面是贪婪重建算法的核心代码函数。

第一步：初始化边界数组映射。

```
HfOld2NewIndex = new int[ Mesh. HalfEdges. Count ];
int count = 0;
foreach ( TriMesh. HalfEdge item in Mesh. HalfEdges)
{
    if ( item. OnBoundary)
    {
        HfOld2NewIndex[ item. Index ] = -1;
    }
    else
    {
        HfOld2NewIndex[ item. Index ] = count;
        count ++ ;
    }
}
```

第二步：初始化堆栈，堆栈中保存要处理的边。

```
Stack < TriMesh. Edge > stack = new Stack < TriMesh. Edge > ( );
bool[ ] VertexFlag    = new bool[ Mesh. Vertices. Count ];
for ( int i = 0;i < VertexFlag. Length;i ++ )
{
    VertexFlag[ i ] = false;
}
bool[ ] Processed = new bool[ Mesh. Faces. Count ];
for ( int i = 0;i < Processed. Length;i ++ )
{
    Processed[ i ] = false;
}
```

第三步：选择三维模型上的一条边 $e^1 = (N_a^1, N_b^1)$。

```
TriMesh. Edge startE = Mesh. Edges[0];
double startLength = (startE. Vertex0. Traits. Position -
                      startE. Vertex1. Traits. Position). Length();
```

第四步：把其中一个顶点确定在二维平面上的位置原点 N_a^1 到(0,0,0)。

```
UnknownU = new double[Mesh. Vertices. Count];
UnknownV = new double[Mesh. Vertices. Count];

UnknownU[startE. Vertex0. Index] =0;
UnknownV[startE. Vertex0. Index] =0;
```

第五步：那么这条边上另一个顶点的位置为 N_b^1 到($\|e^1\|$,0,0)。

```
UnknownU[startE. Vertex1. Index] = startLength;
UnknownV[startE. Vertex1. Index] =0;
```

第六步：把这条边压入堆栈。

```
stack. Push(startE);
```

第七步：假如堆栈不为空，那么弹出一条边 $e = (N_a, N_b)$。

```
while (stack. Count ! =0)
{
    startE = stack. Pop();
```

第八步：对于每个包含此边的面 $f_i = (N_a, N_b, N_c)$，假如此面上所有顶点已经映射过，那么继续弹出下一条边。

```
if (startE. HalfEdge0. Face ! = null &&
    Processed[startE. HalfEdge0. Face. Index] == false)
{
    double beta = newAngle[HfOld2NewIndex[startE. HalfEdge0. Index]];
    double theta = newAngle[HfOld2NewIndex[startE. HalfEdge0. Next. Index]];
    ProcessFace(startE. HalfEdge0, beta, theta, ref UnknownU, ref UnknownV);
    Processed[startE. HalfEdge0. Face. Index] = true;
    stack. Push(startE. HalfEdge0. Next. Edge);
    stack. Push(startE. HalfEdge0. Previous. Edge);
    processedCount ++;
}

if (startE. HalfEdge1. Face ! = null &&
    Processed[startE. HalfEdge1. Face. Index] == false)
{
    double beta = newAngle[HfOld2NewIndex[startE. HalfEdge1. Index]];
    double theta = newAngle[HfOld2NewIndex[startE. HalfEdge1. Next. Index]];
```

```
        ProcessFace(startE. HalfEdge1,beta,theta,ref UnknownU,ref UnknownV);
        Processed[startE. HalfEdge1. Face. Index] = true;
        stack. Push(startE. HalfEdge1. Next. Edge);
        stack. Push(startE. HalfEdge1. Previous. Edge);
        processedCount ++;
    }
```

第九步：假如这个面上的顶点 N_c 还没有被映射，那么根据 N_a、N_b 和角度 α_i^1、α_i^2 和 α_i^3 可以确定此顶点的位置。

```
public void ProcessFace(TriMesh. HalfEdge startHF,double beta,
                        double theta,ref double[] UnknownU,
                        ref double[] UnknownV)
{
    Vector2D v = new Vector2D(UnknownU[startHF. ToVertex. Index] -
                              UnknownU[startHF. FromVertex. Index],
                              UnknownV[startHF. ToVertex. Index] -
                              UnknownV[startHF. FromVertex. Index]);
    Vector2D newV = new Vector2D( -v. y,v. x);
    double tanRatio = Math. Tan(beta)/Math. Tan(theta);
    Vector2D newPosition = Vector2D. Zero;
    beta = beta % (2 * Math. PI);
    theta = theta % (2 * Math. PI);
    if (Math. Tan(beta) >0 && Math. Tan(theta) >0)
    {
        tanRatio = 1/(1 + tanRatio);
        double aPart = v. Length() * tanRatio;
        double hyLength = Math. Tan(beta) * aPart;
        newPosition = tanRatio * v;
        newV = hyLength * newV. Normalize();
    }
    else if (Math. Tan(beta) <0 && Math. Tan(theta) >0)
    {
        tanRatio = 1/( -(ianRatio) -1);
        double aPart = v. Length() * tanRatio;
        double hyLength = -Math. Tan(beta) * aPart;
        newV = hyLength * newV. Normalize();
        newPosition = -tanRatio * v;
    }
    else if (Math. Tan(theta) <0 && Math. Tan(beta) >0)
    {
        tanRatio = 1/(1 + tanRatio);
        double aPart = v. Length() * tanRatio;
```

```
            double hyLength = Math. Tan( beta) * aPart;
            newV = hyLength * newV. Normalize( ) ;
            newPosition = tanRatio * v;
        }
        else if ( Math. Tan( theta) <0 && Math. Tan( beta) <0)
        {
            throw new Exception( "Bad Triangle" ) ;
        }
        newPosition. x = UnknownU[ startHF. FromVertex. Index]
                        + newPosition. x + newV. x;
        newPosition. y = UnknownV[ startHF. FromVertex. Index]
                        + newPosition. y + newV. y;
        UnknownU[ startHF. Next. ToVertex. Index] = newPosition. x;
        UnknownV[ startHF. Next. ToVertex. Index] = newPosition. y;
    }
```

第十步：处理完顶点 N_c 后，设置此面相关标记为已处理。并把(N_a,N_c)和(N_b,N_c)两条边压入堆栈。

```
ProcessFace( startE. HalfEdge1 ,beta ,theta ,ref UnknownU ,ref UnknownV) ;
Processed[ startE. HalfEdge1. Face. Index] = true;
stack. Push( startE. HalfEdge1. Next. Edge) ;
stack. Push( startE. HalfEdge1. Previous. Edge) ;
processedCount ++ ;
```

9.3　最小二乘重建算法

贪婪重建算法的缺点是误差会累计。误差的来源有几个方面：线性 ABF 计算得到的角度本身有误差；在用角度计算顶点坐标时，会有计算误差。从而每个顶点的位置坐标都会有误差，并且后面计算的顶点位置坐标误差会受到前面顶点位置坐标误差的影响。也就是先计算的顶点误差小，后计算的顶点误差大。另一种从角度计算位置的方法是最小二乘（Least Squares）重建算法。这种算法构建一个线性系统，同时计算所有顶点映射后的位置，而不是依次计算每个顶点的位置，因此不会累积误差，误差平均分配到每个顶点上。假如输入的角度不满足每个顶点周围角度之和是 2π，也就是即使输入的角度有误差用最小二乘重建算法也能够得到较好的参数化结果。

1. 数学推导

第一步：对于一个三角形(P_1,P_2,P_3)，那么：

$$\frac{\|\overrightarrow{P_1P_3}\|}{\|\overrightarrow{P_1P_2}\|}=\frac{\sin(\alpha_2^t)}{\sin(\alpha_3^t)} \tag{9-1}$$

第二步：从而可以得出：

$$\overrightarrow{P_1P_3}=\frac{\sin(\alpha_2^t)}{\sin(\alpha_3^t)}\begin{pmatrix}\cos(\alpha_1^t) & \sin(\alpha_1^t)\\ -\sin(\alpha_1^t) & \cos(\alpha_1^t)\end{pmatrix} \tag{9-2}$$

第三步：因此给定一个三角形的两个顶点位置和3个角度，第3个顶点位置就可以从以上公式计算。

第四步：贪婪算法先确定两个顶点位置，其他位置根据上述公式依次进行计算。

第五步：上述公式可以重构为如下形式。

$$\forall t=(j,k,l)\in T,\ M^t(P_j-P_k)+P_l-P_j=0$$

$$M^t=\frac{\sin(\alpha_2^t)}{\sin(\alpha_3^t)}\begin{pmatrix}\cos(\alpha_1^t) & \sin(\alpha_1^t)\\ -\sin(\alpha_1^t) & \cos(\alpha_1^t)\end{pmatrix} \tag{9-3}$$

第六步：那么上述方程构成一个线性系统，未知数为顶点在平面上的 x、y 坐标。线性系统的系数由三角形角度计算出来。

第七步：因为三角形角度确定后，具有缩放和旋转自由度，因此需要确定两个顶点的坐标来去除系统里的自由度，从而得到唯一的解。

第八步：假设 P_{n_v-1} 和 P_{n_v} 到(0,0)和(1,0)，那么去除相关变量后，线性系统公式方程数量是 $2nf$，而未知数变量数量是 $2(n_v-2)$。

第九步：上述系统重构为矩阵形式为：$AP=0$，其中矩阵 A 大小是 $2n_f\times 2(n_v-2)$，而 P 是一个由顶点的 x、y 坐标构成的未知数向量。矩阵 A 里每相邻的两行 A_{2t-1} 和 A_{2t} 由前面的公式计算得出。

第十步：根据欧拉公式，$2n_f$ 大于或等于 $2(n_v-2)$。因此系统需要用最小二乘的方法求解。

2. 算法步骤

第一步：初始化线性系统矩阵。

```
int m = 2 * Mesh. Faces. Count;
int n = 2 * Mesh. Vertices. Count;

SparseMatrix matrix = new SparseMatrix( m,n);
double[ ] bPart = new double[2 * m + 4];
for (int i = 0;i < bPart. Length;i ++ )
{
    bPart[ i] = 0;
}
```

第二步：根据角度计算矩阵元素。

```
TriMesh. Face currentFace = Mesh. Faces[ i];

int p1index = currentFace. GetHalfedge(0). FromVertex. Index;
int p2index = currentFace. GetHalfedge(2). FromVertex. Index;
int p3index = currentFace. GetHalfedge(1). FromVertex. Index;

double a1 = newAngle[ HfOld2NewIndex[ currentFace. GetHalfedge(0). Index]];
```

```
double a2 = newAngle[ HfOld2NewIndex[ currentFace. GetHalfedge(2). Index] ];
double a3 = newAngle[ HfOld2NewIndex[ currentFace. GetHalfedge(1). Index] ];

double m11 = Math. Cos(a1) * Math. Sin(a2)/Math. Sin(a3);
double m12 = Math. Sin(a1) * Math. Sin(a2)/Math. Sin(a3);
double m21 = -Math. Sin(a1) * Math. Sin(a2)/Math. Sin(a3);
double m22 = Math. Cos(a1) * Math. Sin(a2)/Math. Sin(a3);
```

第三步：对于每个三角形，执行如下第四步到第六步。

```
for (int i = 1;i < Mesh. Faces. Count;i ++)
```

第四步：设置和第1个顶点相关的矩阵元素。

```
matrix[2 * currentFace. Index,2 * p1index] = m11 - 1;
matrix[2 * currentFace. Index,2 * p1index + 1] = m12;
matrix[2 * currentFace. Index + 1,2 * p1index] = m21;
matrix[2 * currentFace. Index + 1,2 * p1index + 1] = m22 - 1;
```

第五步：设置和第2个顶点相关的矩阵元素。

```
matrix[2 * currentFace. Index,2 * p2index] = -m11;
matrix[2 * currentFace. Index,2 * p2index + 1] = -m12;
matrix[2 * currentFace. Index + 1,2 * p2index] = -m21;
matrix[2 * currentFace. Index + 1,2 * p2index + 1] = -m22;
```

第六步：设置和第3个顶点相关的矩阵元素。

```
matrix[2 * currentFace. Index,2 * p3index] = 1;
matrix[2 * currentFace. Index + 1,2 * p3index + 1] = 1;
```

第六步：得到两个限制顶点。

```
Vector2D p2 = new Vector2D(0,0);
Vector2D p1 = new Vector2D(100,0);

TriMesh. Face firstFace = Mesh. Faces[0];
int origin1Index = firstFace. GetHalfedge(0). FromVertex. Index;
int origin2Index = firstFace. GetHalfedge(1). FromVertex. Index;
```

第七步：更新系统，添加限制条件。

```
matrix. AddRow();
matrix[ matrix. Rows. Count - 1,2 * origin1Index] = 1;
bPart[ matrix. Rows. Count - 1] = p1. x;

matrix. AddRow();
matrix[ matrix. Rows. Count - 1,2 * origin1Index + 1] = 1;
bPart[ matrix. Rows. Count - 1] = p1. y;
```

```
matrix. AddRow( );
matrix[ matrix. Rows. Count - 1 ,2 * origin2Index ] = 1;
bPart[ matrix. Rows. Count - 1 ] = p2. x;

matrix. AddRow( );
matrix[ matrix. Rows. Count - 1 ,2 * origin2Index + 1 ] = 1;
bPart[ matrix. Rows. Count - 1 ] = p2. y;
```

第八步：求解线性方程组。

```
double[ ] x = LinearSystem. Instance. SloveLeastSquare( ref matrix ,ref bPart) ;
```

第九步：得到最终顶点位置坐标。

```
UnknownU = new double[ Mesh. Vertices. Count ];
UnknownV = new double[ Mesh. Vertices. Count ];

for ( int i = 0;i < Mesh. Vertices. Count;i ++ )
{
    int index = Mesh. Vertices[ i ]. Index;
    UnknownU[ index ] = x[ 2 * index ];
    UnknownV[ index ] = x[ 2 * index + 1 ];
}
```

9.4 两种重建效果分析

最小二乘重建算法由于所有顶点一起计算，而不是像贪婪重建算法一样依次计算每个顶点，因此大多说情况下，最小二乘重建算法效果比贪婪重建算法好。本节用几个例子来展示两种算法的对比。

（1）可展开曲面（Developable Surface）是指高斯曲率为零的曲面，可以没有任何扭曲地展开为平面。对于可扩展曲面来说，贪婪重建算法和最小二次重建算法的结果没有很大差别。如图 9-1 所示，圆柱和圆锥的参数化展开，用两种算法得到的结果类似，都可以展开为一个平面。在圆锥的例子上，贪婪重建算法得到的效果更好。

（2）对于近似于平面的三维模型，如图 9-2 所示的面具三维模型，两种算法都可以成功展开，但最小二乘重建算法得到的效果更好。并且在角度、扭曲、面积等各种度量指标里面，最小二乘重建算法得到的扭曲也比贪婪重建算法得到的扭曲小。

（3）对于和平面相差较远的三维模型，如兔子头，贪婪重建算法展开后会出现面翻转的情况。如图 9-3 所示，橘红色的部分表示翻转的面。而最小二乘重建算法能够成功重建。从中可以看出，最小二乘法重建算法相比贪婪重建算法来说，能够在更多的模型上成功使用。

（4）对于有洞的三维模型，最小二乘重建算法可以成功展开，而贪婪重建算法就展开失败。如图 9-4 所示的汽车模型，身上有几个洞，在经过最小二乘重建算法之后，可以展开为一个平面。而贪婪重建算法就无法处理这种模型了。

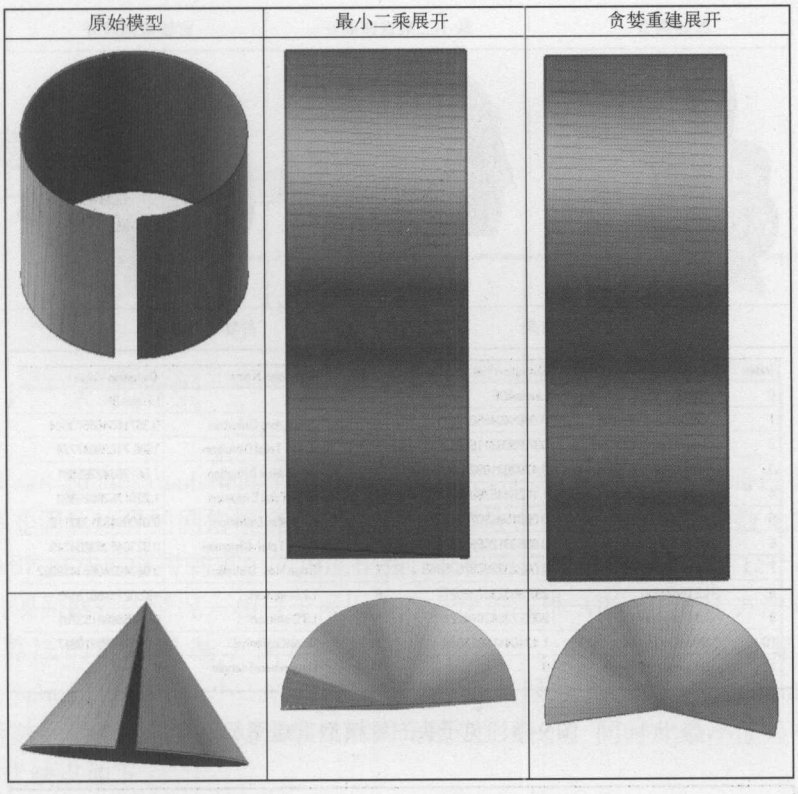
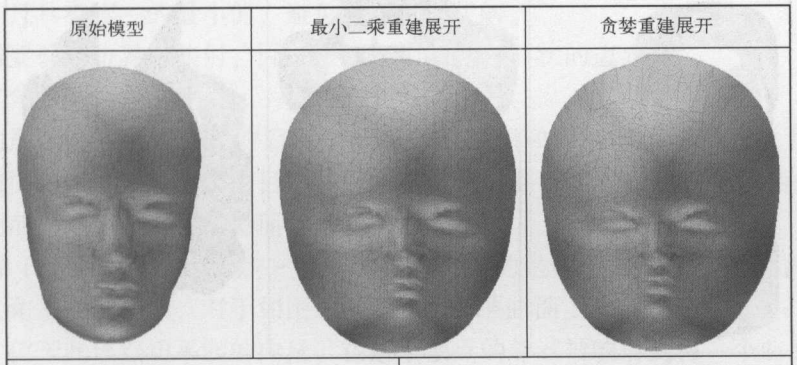


图 9-1　可扩展三维模型重建图示



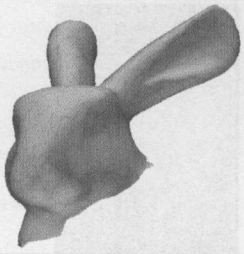
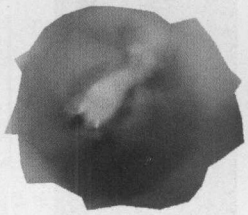
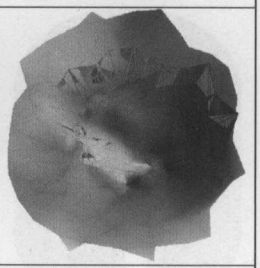
最小二乘重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.0186397390123363
2	Angle Total Distortion	129.620745091787
3	Angle Max Distortion	0.520344374695084
4	Area Total Distortion	0.23861007718104
5	Area Max Distortion	0.000790667097675662
6	Edge Total Distortion	0.131740769693061
7	Edge Max Distortion	0.000211316290137927
8	L2 Distortion	1.08415927994138
9	L8 Distortion	20.2100835941924
10	QuasiConformal	1.04343855435886
11	DegenerateTriangle	0

贪婪重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.129322386108577
2	Angle Total Distortion	899.307872999042
3	Angle Max Distortion	2.0548064239555
4	Area Total Distortion	0.464863401693381
5	Area Max Distortion	0.00318023614832387
6	Edge Total Distortion	0.2442619069815
7	Edge Max Distortion	0.00090079005341289 5
8	L2 Distortion	29.2997334122325
9	L8 Distortion	1533.86637071649
10	QuasiConformal	2.98999673492677
11	DegenerateTriangle	0

图 9-2　近似于平面三维模型重建图示

原始模型	最小二乘重建展开	贪婪重建展开
		

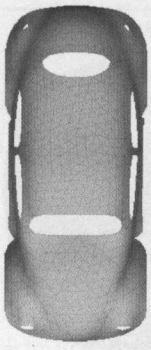
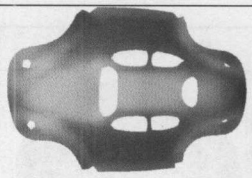
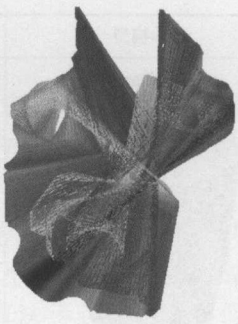
最小二乘重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.088640455019977
2	Angle Total Distortion	387.890631167419
3	Angle Max Distortion	0.470202109244637
4	Area Total Distortion	1.16264457681676
5	Area Max Distortion	0.00815862074949225
6	Edge Total Distortion	0.856381268911941
7	Edge Max Distortion	0.00223229737675468
8	L2 Distortion	297.903233728027
9	L8 Distortion	2005.73843944171
10	QuasiConformal	1.42464418612549
11	DegenerateTriangle	0

贪婪重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.367165468573524
2	Angle Total Distortion	1606.71609047774
3	Angle Max Distortion	2.66136842265581
4	Area Total Distortion	1.22817626626992
5	Area Max Distortion	0.010894831393112
6	Edge Total Distortion	0.923845353284749
7	Edge Max Distortion	0.00240340683418062
8	L2 Distortion	482.022545819185
9	L8 Distortion	6449.58688153781
10	QuasiConformal	7.80920275916847
11	DegenerateTriangle	0

图 9-3　兔子头三维模型重建图示

原始模型	最小二乘重建展开	贪婪重建展开
		

最小二乘重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.0130713214380721
2	Angle Total Distortion	71.9445531951489
3	Angle Max Distortion	0.323835085614546
4	Area Total Distortion	0.271669751554947
5	Area Max Distortion	0.000508482567939207
6	Edge Total Distortion	0.141369097648821
7	Edge Max Distortion	0.000174135043127796
8	L2 Distortion	1.06004263051587
9	L8 Distortion	1.38658253009191
10	QuasiConformal	1.02270254031179
11	DegenerateTriangle	0

贪婪重建结果

Index	Distortion Name	Distortion Value
0	Name	LinearABF
1	Angle Avg Distortion	0.496284162188978
2	Angle Total Distortion	2731.54802868813
3	Angle Max Distortion	2.49944751967408
4	Area Total Distortion	1.13738948103112
5	Area Max Distortion	0.0161255423813855
6	Edge Total Distortion	1.2177518787117
7	Edge Max Distortion	0.00275862189135156
8	L2 Distortion	163.471680536866
9	L8 Distortion	8313.49652302035
10	QuasiConformal	72.0942054546309
11	DegenerateTriangle	0

图 9-4　有洞的多边界三维模型重建图示

第 10 章 LSCM 算法

10.1 保角映射

LSCM（Least Square Conformal Maps）是指最小二乘保角映射。可以通过不添加一层虚拟边界，处理非固定边界的网格模型参数化。这个算法也是线性的，但参数化的结果相对于前述的线性 ABF 方法能够得到更小的角度扭曲，在保角方面的效果更好。LSCM 在有钝角的情况下，会出现面翻转的情况。相对于采用重建方法的线性 ABF，LSCM 方法效率和可靠性更高。LSCM 算法的原理是基于保角映射的数学定义。LSCM 数学基础是柯西 - 黎曼（Cauchy - Riemann）方程的最小二乘近似，是一种近似保角（Quasi - Conformal）算法。所采用的目标函数的最小化值可以使参数化后的角度变形最小，同时此最小值是唯一的。

LSCM 的优缺点如下。

（1）减少了角度扭曲和不一致的缩放。

（2）存在且具有唯一的最小值，避免了局部最优解。

（3）不需要确定边界的映射，因此可以作用于任意形状的边界。

（4）有三角形翻转和重叠。

参数化的最好的情况是映射前后每个三角形的面积和角度都不变，但除了可展开曲面外，其他的曲面都无法满足这样的目标。因此参数化的目标大部分就定义为保持角度不变，在角度不变的前提下，尽可能保持面积不变。

保角映射（Conformal Map）是一个数学概念，指的是两个曲面映射后和前每一个点上两个向量的夹角不变。例如，对于如图 10-1 所示三维曲面上的一个顶点(u,v)，任意两个互相垂直的、位于此顶点切平面的向量，在这个顶点的邻域被映射到另一个曲面上时，原来曲面上的两个互相垂直的向量映射后依然互相垂直。

也就是如果把平面域(u,v)映射到曲面是保角的，那么通过$X(u,v)$的两个方向的切线正交，并且范式相等，也就是满足如下的柯西 - 黎曼方程：

$$N(u,v) \times \frac{\partial X}{\partial u}(u,v) = \frac{\partial X}{\partial v}(u,v) \tag{10-1}$$

保角映射是局部各向同性（Local Isotropic）的，也就是把一个局部的小圆映射到曲面上的小圆。柯西 - 黎曼方程表示的是光滑的曲面的保角映射，而三维网格的映射需要对上述方程做离散化处理。黎曼定理指出如果两个曲面和圆盘拓扑同构，那么这两个曲面之间一定存在一个保角的映射。在光滑的情况下，原来直线在保角映射后，并不能保持直线。但是在离散的情况下，如果需要把三角形的边保持直线不变，并且每个三角形内，映射是线性的，那

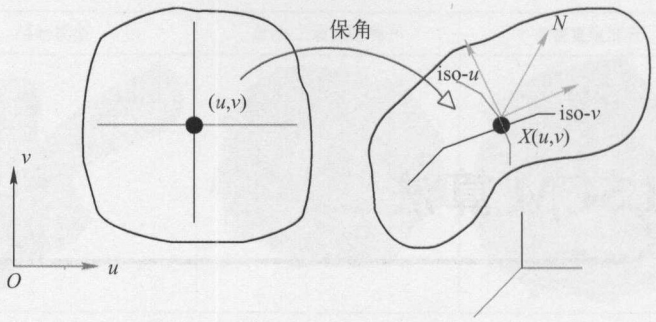


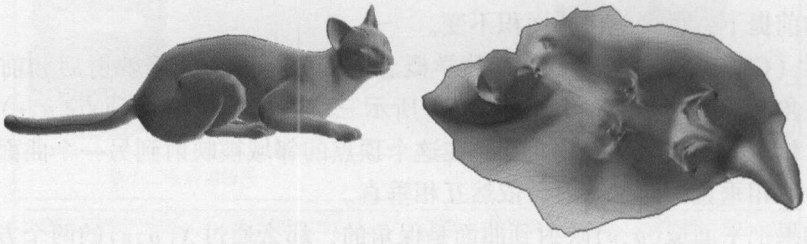
图 10-1　保角映射

么黎曼定理在离散情况下就不能成立。

LSCM 算法建立在柯西 - 黎曼方程的基础上。首先需要定义一个能量函数。在定义的保角能量函数是和每个三角形面积成正比的，假如三角形面积减少，此能量也减少，因此会造成退化解，所以需要确定两个顶点的位置，从而最终的参数化结果受到选择的两个顶点的影响。ABF 定义的能量函数和面积没有关系，直接作用在角度上，因此 LSCM 能得到更好的效果。

在定义了能量函数之后，因为三维模型不是光滑的曲面，而是离散的，因此还需要进行离散化。在对柯西 - 黎曼方程和能量函数进行离散化处理时，如果在顶点上进行离散化，那么柯西 - 黎曼方程可以用余切公式写为两个独立的线性系统。由于 u、v 之间没有直接关系，因此需要确定一个边界的映射，也就是和谐映射。如果不在顶点上进行离散化，而是在三角形面上进行离散化，考虑面的保角映射，那么保角映射可以构造为一个无约束的二次方最小化问题。u 和 v 两个参数由一个单独的全局方程联系起来，因此可以处理复杂的边界。也就是与和谐参数化算法不一样，和谐参数化是 u、v 坐标互不影响，独立计算。而 LSCM 方法把 u、v 未知变量放到一个系统里面，从而 u、v 之间互相约束。因为 u、v 之间互相约束，从而可以减少边界的约束。

LSCM 参数化图示如图 10-2 所示。



（a）三维模型　　（b）LSCM参数化

图 10-2　LSCM 参数化图示

10.2　保角离散化

定义在光滑曲面上的能量函数需要进行离散化。通过离散化构建一个线性的系统，这个线性系统的思路是把 u、v 坐标表示为一个复数。离散的步骤如下。

第一步：假设三维模型的每个三角形有一个局部正交坐标系，那么三角形的 3 个顶点的局部坐标为(x_1,y_1)、(x_2,y_2)、(x_3,y_3)，法向是沿 Z 轴方向。相邻两个三角形的方向一致。

第二步：假设 $U:(x,y)\rightarrow(u,v)$ 是从三维空间到二维平面的映射，那么在局部坐标系里，柯西 - 黎曼方程离散化为：

$$\frac{\partial X}{\partial u}-i\frac{\partial X}{\partial u}=0 \tag{10-2}$$

第三步：如果 X 用复数表示为 $X=x+iy$，让 $U=u+iv$。那么根据反函数导数定理得出：

$$\frac{\partial U}{\partial x}+i\frac{\partial U}{\partial y}=0 \tag{10-3}$$

第四步：由于此公式不能被完全满足，因此可以用最小平方的形式满足。

第五步：用 A_T 表示三角形面积，从而对于每个三角形定义下面的能量函数，保角参数化就是求取此能量函数的最小值：

$$C(T)=\int_T\left|\frac{\partial U}{\partial x}+i\frac{\partial U}{\partial y}\right|^2\mathrm{d}A=\left|\frac{\partial U}{\partial x}+i\frac{\partial U}{\partial y}\right|^2A_T \tag{10-4}$$

第六步：对于所有三角形来说，能量函数为所有单个三角形能量函数之和：

$$C(T)=\sum_{T\in T}C(T) \tag{10-5}$$

第七步：进而为每个顶点设置一个复数 U_3，从而使柯西 - 黎曼方程子最小平方的情况下满足。

第八步：假设 U 在三角形内部是线性变化的，对于一个三角形 $\{(x_1,y_1),(x_2,y_2),(x_3,y_3)\}$ 来说，那么 3 个顶点上的值 u_1、u_2、u_3 有如下关系：

$$\begin{pmatrix}\partial u/\partial x\\ \partial u/\partial y\end{pmatrix}=\frac{1}{d_T}\begin{pmatrix}y_2-y_3 & y_3-y_1 & y_1-y_2\\ x_3-x_2 & x_1-x_3 & x_2-x_1\end{pmatrix}\begin{pmatrix}u_1\\ u_2\\ u_3\end{pmatrix} \tag{10-6}$$

第九步：其中 $d_T=(x_1y_2-y_1x_2)+(x_2y_3-y_2x_3)+(x_3y_1-y_3x_1)$ 是三角形面积的两倍。

第十步：把上面的向量形式改为复数形式为：

$$\frac{\partial U}{\partial x}+i\frac{\partial U}{\partial y}=\frac{i}{d_T}(W_1\quad W_2\quad W_3)(u_1\quad u_2\quad u_3)^{\mathrm{T}} \tag{10-7}$$

第十一步：其中

$$\begin{cases}W_1=(x_3-x_2)+i(y_3-y_2)\\ W_2=(x_1-x_3)+i(y_1-y_3)\\ W_3=(x_2-x_1)+i(y_2-y_1)\end{cases} \tag{10-8}$$

第十二步：表示为复数 $U_j=u_j+iv_j$，最终柯西 - 黎曼方程可以变化为：

$$\frac{\partial U}{\partial x}+i\frac{\partial U}{\partial y}=\frac{i}{d_T}(W_1\quad W_2\quad W_3)(U_1\quad U_2\quad U_3)^{\mathrm{T}} \tag{10-9}$$

第十三步：那么能量函数改变变量后变为：

$$C(U=(U_1,\cdots,U_n)^{\mathrm{T}})=\sum_{T\in T}C(T)$$

第十四步：也就是每个三角形的能量函数为：

$$C(T)=\frac{1}{d_T}|(W_{j1,T}\quad W_{j2,T}\quad W_{j3,T})(U_{j1}\quad U_{j2}\quad U_{j3})^{\mathrm{T}}|^2 \tag{10-10}$$

第十五步：能量函数 $C(U)$ 是复数 $U_1,\cdots,U_n$ 的二次形式，用矩阵形式表示为：

$$C(U)=U^*CU \tag{10-11}$$

第十六步：其中 C 是 $n\times n$ 的 Hermitia 对称矩阵。U^* 表示 U 的哈密尔顿共轭矩阵（Conjugate）。C 是哈密尔顿 Gram 矩阵，可以写为：

$$C=M^*M \tag{10-12}$$

第十七步：上式中的矩阵 $M=(m_{ij})$ 是大小为 $F\times V$ 的矩阵，其中

$$m_{ij}=\begin{cases}\dfrac{W_{j,T_i}}{\sqrt{d_{T_i}}} & \text{顶点 } j \text{ 属于三角形 } T\\ 0 & \text{其他}\end{cases} \tag{10-13}$$

第十八步：如果没有额外的约束条件，上述优化问题的解都是 0。为了使优化问题避免简单的解，那么需要确定一些点的位置作为约束。

第十九步：如果确定 p 个点，那么 $U=(U_f^{\mathrm{T}},U_p^{\mathrm{T}})^{\mathrm{T}}$。

第二十步：也就是矩阵 $M=(m_{ij})$ 分为两部分

$$M=(M_f \quad M_p)$$

式中，M_f 是大小为 $n'\times(n-p)$ 的矩阵，M_p 是 $n'\times p$ 矩阵。

第二十一步：从而前面的方程变为：

$$C(U)=U^*M^*MU=\|MU\|^2=\|M_fU_f+M_pU_p\|^2 \tag{10-14}$$

第二十二步：最终得出线性系统为：

$$C(x)=\|Ax-b\|^2 \tag{10-15}$$

第二十三步：线性系统的中左边的矩阵和右边的向量分别为：

$$\begin{aligned}A&=\begin{pmatrix}M_f^1 & -M_f^2\\ M_f^2 & M_f^1\end{pmatrix}\\ b&=-\begin{pmatrix}M_p^1 & -M_p^2\\ M_p^2 & M_p^1\end{pmatrix}\begin{pmatrix}U_p^1\\ U_p^2\end{pmatrix}\end{aligned} \tag{10-16}$$

第二十四步：这个线性系统具有如下的特性。

（1）A 在约束点大于等于 2 的情况下，是满秩的。

（2）如果约束点等于 2，那么 $x=(A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}b$，假如曲面是可扩展面，那么映射是完全保角的，也就是目标函数的最小值是零。

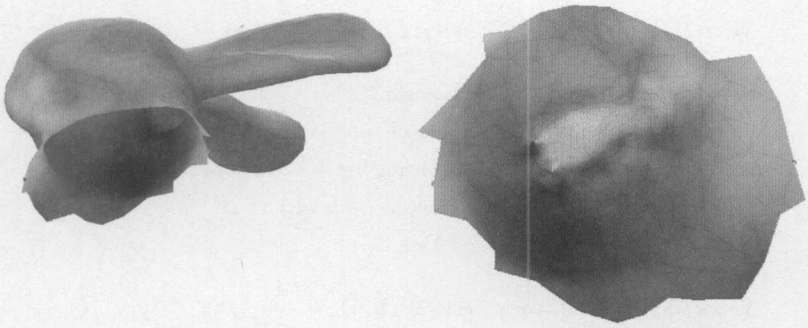
（3）最小值是旋转不变的。

（4）最小值和模型的分辨率无关。

10.3 LSCM 算法步骤

1. 固定点选择

LSCM 算法需要预先固定两个顶点作为系统的限制约束条件，否则就得到一个都是 0 的解。通常这两个顶点选择边界上相距最远的两个顶点。如图 10-3 所示的三维模型和参数化后的二维模型，其中红色和蓝色的点就是边界上两个固定的顶点。



（a）原始模型　　（b）LSCM参数化

图 10-3　固定点选择

选择固定点可以手工选择，也可以通过边界取得第一个点和最中间的点，代码如下。

```
public static List < TriMesh. Vertex > RetrieveTwoFixBoundaryPoint( TriMesh mesh)
{
    List < TriMesh. Vertex > two = new List < HalfEdgeMesh. Vertex > ( ) ;
    List < TriMesh. Vertex > bound = TriMeshUtil. RetrieveBoundarySingle( mesh) ;
    foreach ( TriMesh. Vertex vertex in bound)
    {
        if ( vertex. Traits. SelectedFlag == 1)
        {
            two. Add( vertex) ;
        }
    }
    if ( two. Count < 2)
    {
        two. Add( bound[ 0] ) ;
        two. Add( bound[ bound. Count/2] ) ;
    }
    return two;
}
```

2. 算法步骤

虽然 LSCM 的推导过程很复杂，但最终得到的是一个线性系统。因此，整个算法的核心就是构建线性系统的左边矩阵和右边的向量。而左边矩阵又分为 4 个模块。整个算法步骤如下。

第一步：初始化边长向量，用于之后构建矩阵。

```
public Vector2D[ ][ ] CalculateEdgeVectors( TriMesh Mesh )
{
    Vector2D[ ][ ] vectors = new Vector2D[ Mesh. Faces. Count][ ];
    foreach ( TriMesh. Face face in Mesh. Faces)
    {
        Vector3D vertex1 = face. GetVertex( 0) . Traits. Position;
        Vector3D vertex2 = face. GetVertex( 1) . Traits. Position;
```

```
            Vector3D vertex3 = face.GetVertex(2).Traits.Position;

            double a = (vertex1 - vertex2).Length();
            double b = (vertex2 - vertex3).Length();
            double c = (vertex3 - vertex1).Length();
            double angle = Math.Acos((a * a + c * c - b * b)
                                     / (2 * a * c));

            Vector2D UPosition1 = new Vector2D(0,0);
            Vector2D UPosition2 = new Vector2D(a,0);
            Vector2D UPosition3 = new Vector2D(c * Math.Cos(angle),
                                               c * Math.Sin(angle));

            Vector2D[] triVectors = new Vector2D[3];
            Vector2D EdgeVectorsItem1 = UPosition3 - UPosition2;
            Vector2D EdgeVectorsItem2 = UPosition1 - UPosition3;
            Vector2D EdgeVectorsItem3 = UPosition2 - UPosition1;

            triVectors[0] = EdgeVectorsItem1;
            triVectors[1] = EdgeVectorsItem2;
            triVectors[2] = EdgeVectorsItem3;
            vectors[face.Index] = triVectors;
        }
        return vectors;
    }
```

第二步：得到位于边界上两个顶点，这两个顶点作为系统的限制条件。

```
boundary = new ParaBoundary(Mesh);
boundary.GetFixTwoPoint();
```

第三步：矩阵构建时，需要把两个固定的顶点顺序放到矩阵后面，因此需要做顶点顺序映射。分别定义两个数组保持正向和反向映射的序号。

```
public int[] VertexMapping;
public int[] antiVertexMapping;
```

第四步：设定两个固定点的映射序号。

```
VertexMapping = new int[Mesh.Vertices.Count];
antiVertexMapping = new int[Mesh.Vertices.Count];
VertexMapping[boundary.fixVertexOne.Index] = VertexMapping.Length - 2;
VertexMapping[boundary.fixVertexTwo.Index] = VertexMapping.Length - 1;
antiVertexMapping[VertexMapping.Length - 2] = boundary.fixVertexOne.Index;
antiVertexMapping[VertexMapping.Length - 1] = boundary.fixVertexTwo.Index;
```

第五步：设定其他顶点的正向反向映射序号。

```
int itrCount = 0;
foreach (TriMesh. Vertex v in Mesh. Vertices)
{
    if (v. Index == boundary. fixVertexTwo. Index
        || v. Index == boundary. fixVertexOne. Index)
    {
        continue;
    }
    VertexMapping[v. Index] = itrCount;
    antiVertexMapping[itrCount] = v. Index;
    itrCount ++;
}
```

第六步：定义 4 个稀疏基本矩阵。

```
public SparseMatrix realMf;
public SparseMatrix realMp;
public SparseMatrix imageMf;
public SparseMatrix imageMp;
```

第七步：初始化 4 个基本矩阵。

```
int faceCount = Mesh. Faces. Count;
int VertexCount = Mesh. Vertices. Count;
realMf = new SparseMatrix(faceCount, VertexCount - 2);
imageMf = new SparseMatrix(faceCount, VertexCount - 2);
realMp = new SparseMatrix(faceCount, 2);
imageMp = new SparseMatrix(faceCount, 2);
```

第八步：对于所有三角形，计算 4 个矩阵的相应元素。

```
Vector2D[] triEdgeVectors = edgeVectors[triangle. Index];
for (int i = 0; i < 3; i ++)
{
    faceVerties[i] = triangle. GetVertex(i);
}
double area = TriMeshUtil. ComputeAreaFace(triangle);
double denominator = Math. Sqrt(2 * area);
for (int i = 0; i < 3; i ++)
{
    TriMesh. Vertex v = faceVerties[i];
    Vector2D edgeVector = triEdgeVectors[i];
```

第九步：对于每个三角形，计算两个实矩阵的相应的项。

```
double dX = edgeVector. x;
if (v. Index ! = boundary. fixVertexOne. Index && v. Index
    ! = boundary. fixVertexTwo. Index)
```

```
    {
        int newMappingIndex = VertexMapping[v.Index];
        realMf[triangle.Index,newMappingIndex] = dX/denominator;
    }
    else if (v.Index == boundary.fixVertexOne.Index)
    {
        realMp[triangle.Index,0] = dX/denominator;
    }
    else if (v.Index == boundary.fixVertexTwo.Index)
    {
        realMp[triangle.Index,1] = dX/denominator;
    }
```

第十步：对于每个三角形，计算两个虚矩阵的相应的项。

```
double dY = edgeVector.y;
if (v.Index != boundary.fixVertexOne.Index && v.Index
    != boundary.fixVertexTwo.Index)
{
    int newMappingIndex = VertexMapping[v.Index];
    imageMf[triangle.Index,newMappingIndex] = dY/denominator;
}
else if (v.Index == boundary.fixVertexOne.Index)
{
    imageMp[triangle.Index,0] = dY/denominator;
}
else if (v.Index == boundary.fixVertexTwo.Index)
{
    imageMp[triangle.Index,1] = dY/denominator;
}
```

第十一步：得到4个基本矩阵支护，可以根据如下形式，构建线性系统矩阵。

```
A = |realMf, -imageMf|
    |imageMf,realMf|
public SparseMatrix BuildMatrixA()
{
    SparseMatrix A = new SparseMatrix(2 * Mesh.Faces.Count,
                                      2 * Mesh.Vertices.Count - 4);

    int FaceCount = Mesh.Faces.Count;
    int VerticesCount = Mesh.Vertices.Count;
    int faceOffset = FaceCount;
    int verticesOffset = VerticesCount - 2;
    for (int i = 0;i < FaceCount;i ++)
    {
```

```
            for (int j = 0;j < VerticesCount - 2;j ++)
            {
                double realItemValue = realMf[i,j];
                double imageItemValue = imageMf[i,j];
                if (realItemValue ! = 0)
                {
                    A[i,j] = realItemValue;
                    A[faceOffset + i,verticesOffset + j] = realItemValue;
                }
                if (imageItemValue ! = 0)
                {
                    A[i,verticesOffset + j] = - imageItemValue;
                    A[faceOffset + i,j] = imageItemValue;
                }
            }
        }
        return A;
    }
```

第十二步：设置好两个固定顶点在二维平面的位置。

```
    private double[] FixPinPosition()
    {
        double[] U = new double[4];
        U[0] = boundary. handleFirst. x;
        U[1] = boundary. handleSecond. x;
        U[2] = boundary. handleFirst. y;
        U[3] = boundary. handleSecond. y;
        return U;
    }
```

第十三步：根据 4 个基本矩阵，构建用于计算方程右边向量的矩阵。

```
    private SparseMatrix BuildMatrixBM()
    {
        SparseMatrix BM = new SparseMatrix(2 * Mesh. Faces. Count,4);
        int FaceCount = Mesh. Faces. Count;
        int VerticesCount = 2;
        int faceOffset = FaceCount;
        int verticesOffset = VerticesCount;
        for (int i = 0;i < FaceCount;i ++)
        {
            for (int j = 0;j < VerticesCount;j ++)
            {
                double realItemValue = realMp[i,j];
                double imageItemValue = imageMp[i,j];
```

```
                if (realItemValue != 0)
                {
                    BM[i,j] = -realItemValue;
                    BM[faceOffset + i, verticesOffset + j] = -realItemValue;
                }
                if (imageItemValue != 0)
                {
                    BM[i, verticesOffset + j] = imageItemValue;
                    BM[faceOffset + i,j] = -imageItemValue;
                }
            }
        }
        return BM;
    }
```

第十四步：构建方程的右边向量。

```
    public double[] BuildRightB(double[] pinPosition)
    {
        SparseMatrix BM = BuildMatrixBM();
        double[] bPart = BM.Multiply(pinPosition);
        return bPart;
    }
```

第十五步：求解线性方程组。

```
    public override void Parameterize()
    {
        Vector2D[][] edgeVectors = CalculateEdgeVectors(Mesh);
        Init();
        BuildBasicMatrix(edgeVectors);
        SparseMatrix A = BuildMatrixA();
        double[] pinPosition = FixPinPosition();
        double[] bPart = BuildRightB(pinPosition);
        double[] newP = LinearSystem.Instance.SloveLeastSquare(ref A, ref bPart);
        MapBackUV(newP, pinPosition);
    }
```

第十六步：把得到的未知数重新映射到原来的顺序。

```
    public void MapBackUV(double[] UV, double[] pinPosition)
    {
        UnknownU = new double[Mesh.Vertices.Count];
        UnknownV = new double[Mesh.Vertices.Count];
        int Pin1Index = antiVertexMapping[VertexMapping.Length - 2];
```

```
        int Pin2Index = antiVertexMapping[ VertexMapping. Length - 1 ];
        UnknownU[ Pin1Index ] = pinPosition[ 0 ];
        UnknownV[ Pin1Index ] = pinPosition[ 2 ];
        UnknownU[ Pin2Index ] = pinPosition[ 1 ];
        UnknownV[ Pin2Index ] = pinPosition[ 3 ];
        int vertexOffset = Mesh. Vertices. Count - 2;
        for ( int i = 0; i < VertexMapping. Length - 2; i ++ )
        {
            int index = antiVertexMapping[ i ];
            double positionX = UV[ i ];
            double positionY = UV[ vertexOffset + i ];
            UnknownU[ index ] = positionX;
            UnknownV[ index ] = positionY;
        }
    }
```

10.4　LSCM 实验效果分析

LSCM 算法参数化能够处理不同分辨率的模型，并得到同样的结果。如图 10-4 所示的

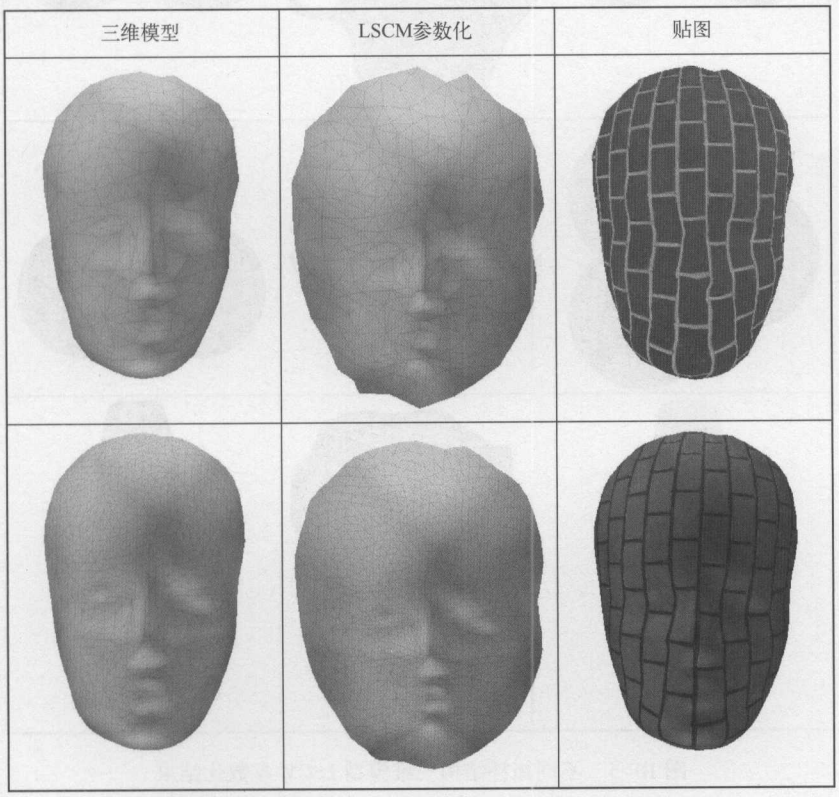


图 10-4　不同分辨率三维模型 LSCM 参数化结果

面具三维模型所示。第一行是面数的面数为 500，第二行面数为 2000，在经过参数化贴图后，得到的贴图结果是一致的。

LSCM 对于各种拓扑的模型在剪开后，也能够成功地进行参数化和贴图。如图 10-5 所示，对于亏格分别为 1、2、3 的圆环进行参数化展开和贴图。从这些实验中可以看出，相对于线性 ABF，LSCM 能够处理更多的三维模型。这主要是由于 LSCM 是直接构建在数学系统上的，具有坚实的数学理论基础。并且在离散化时离散的是数学模型，也就是以光滑情况下的数学概念为指导，构建了一个离散的数学模型。这样可以更好地概括包含保角映射的属性，从而得到一个更好的参数化展开结果。

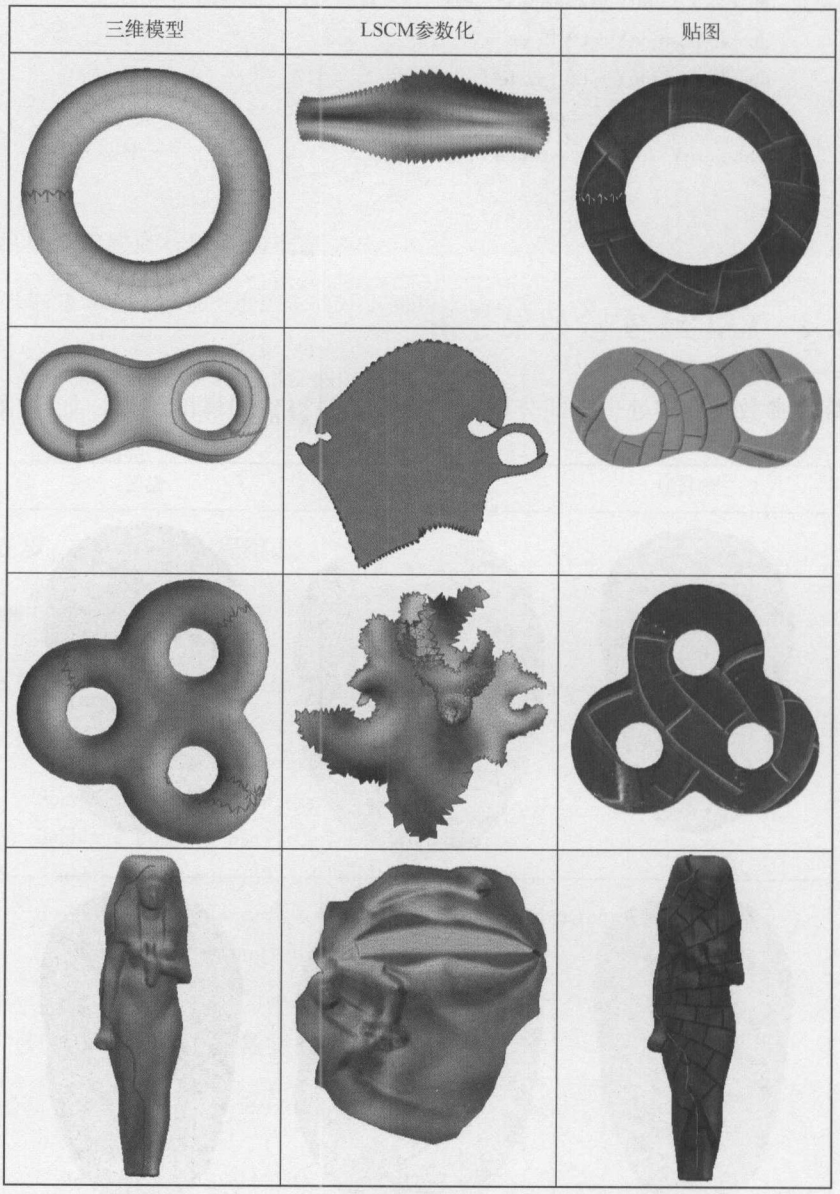


图 10-5　不同拓扑结构三维模型 LSCM 参数化结果

约束点参数化。很多时候，需要限制三维模型上特定的点在参数化后的二维位置。例如，对于一个人脸模型，需要把眼睛的顶点固定到平面上相应的位置。可以在我们的线性系统中去掉相应的顶点对应的方程，如去掉 m 个约束点，得到一个$(2V-4-2m)\times(2V-4-2m)$的线性系统。

或者加上用软约束的方式，在原有的系统上加上相应的有权重方程，也就是得到一个$(2V-4+2m)\times(2V-4)$的线性系统，这个系统可以用最小二次方的方法求解。

第11章 DCP和DAP参数化算法

11.1 内在参数化概念

1. 定义

把三维模型映射到平面上的理论工具有很多，离散的保角参数化可以从微分几何、有限元等数学理论推导出来。LSCM从全局的角度进行数学模型构建，也就是把三维模型当作一个整理来构建数学系统。也可以从局部和内在的角度来观察三维模型，从而构建数学系统。三维模型位于欧几里得三维空间，但三维模型本身是个二维的曲面，具有一些内在（Intrinsic）的属性。从三维模型内在的属性出发，进行参数化数学模型的构建，称为内在参数化（Intrinsic Parameterization）。

三维模型映射到二维平面，如果角度保持不变（Angle - preserving），称为保角（Conformal）映射。如果保持面积不变（Area - preserving），称为等积映射（Authalic）。但任何映射都无法既保持角度不变，还保持面积不变。也就是无法保持边长不变（Distance - preserving）。保持边长不变的映射称为等容积的（Isometric）。

内在参数化算法中保持角度不变的参数化称为DCP（Discrete Conformal Parameterization）。保持面积不变的参数化称为DAP（Discrete Authalic Parameterization）。DCP和DAP参数化方法也分为固定和不固定边界两种。相对于固定边界的和谐参数化来说，不固定边界的内在参数化算法方法的参数化结果扭曲程度较小，尤其在映射的边界附近。但这种方法不保证能得到双射或局部双射的结果，可能造成三角形翻转和全局重叠情况。

如图11-1所示是DCP和DAP参数化算法在固定边界和不固定边界的参数化结果。从中可以看出，不固定边界的DCP和LSCM参数化算法得到的展开平面类似。

（a）三维模型　　（b）固定边界　　（c）不固定边界

图11-1　DCP参数化算法图示

给定一个三维网格模型 M_1 和一个二维平面三角形网格 $u \in R^2$，如果

$$x_i = (x_i, y_i, z_i)^t,$$

$$u_i = (u_i, v_i)^t$$

分别表示三维网格和二维平面上的顶点，那么参数化可以表示为一个分段线性映射：

$$\psi : M \rightarrow u$$

$$x_i \rightarrow u_i$$

内在参数化是另一种参数化方法，它的参数化结果的目标和之前不同。内在参数化的目标是：三维网格模型 M 映射到一个二维平面三角形网格 $u \in R^2$ 后，此映射能够保持原来 M 的内在属性。

2. 度量函数

对于一个三维模型 M，用 ϕ 表示网格模型的一个度量函数。一个度量函数 ϕ 必须满足如下条件。

（1）旋转和位移不变。

（2）连续性，也就是离散模型在逼近光滑时候连续，即：

$$\phi(M_n) \rightarrow \phi(M) \text{ 如果 } M_n \rightarrow M, n \rightarrow \infty \tag{11-1}$$

（3）满足加法交换律：

$$\phi(M_1 \cup M_2) + \phi(M_1 \cap M_2) = \phi(M_1) + \phi(M_2) \tag{11-2}$$

3. 内在度量函数

一个内在属性的度量是指度量函数只依赖于三维模型本身，和三维模型的采样等无关。例如，在三维模型的边界或内部三角形上添加顶点，并不改变三角形的内在属性。也就是三维模型添加顶点后的几何属性并没有改变，改变的只是离散化的结果。

对于三维网格模型来说，满足上述条件的内在属性如下。

（1）面积。

（2）周长。

（3）欧拉示性数。

11.2　内在参数化能量函数

和其他参数化方法一样，内在参数化算法也需要定义一个能量函数，求得此能量函数的最小值就得到参数化的结果。内在参数化的能量函数是定义在一个顶点和此顶点的一层领域三角形上的，每个顶点邻域上的能量函数之和就构成了整个三维模型上的能量函数。

原来的曲面和参数化映射后的曲面的能量函数定义如下。

$$E : T \times T \rightarrow R$$

$$(M, u) \rightarrow E(M, u)$$

这个能量函数有如下特点：因为映射到自身的扭曲最小，所以对于任意的三维模型，映射到自身得到的能量函数比映射到其他曲面得到的能量函数都要小。

$$E(M, M) \leqslant E(M, u) \tag{11-3}$$

为了方便，可以把映射到自身的能量函数表示为三维模型的一个内在函数 ϕ，也就是：

$$E(M, M) = \phi(M) \tag{11-4}$$

在得到能量函数后，此能量函数的未知数是参数化后的坐标位置，因此求导可以得到此能量函数的最小值。因为此能量函数是个内在度量函数，所以这个最小值够保持内在属性扭曲的最小。能量函数求导如下。

$$\frac{\partial E}{\partial u_i}=0 \tag{11-5}$$

能量函数是度量三维模型参数化映射前和映射后的函数。内在度量函数是度量模型本身的函数。通过把能量函数和内在度量函数关联起来，就可以把三维模型内在的属性容纳到能量函数里面，从而使能量函数的设计能够在参数化后保持内在的属性不变。从中可以看出，虽然参数化算法的设计大多数需要构建一个能量函数，但这个能量函数各有不同，需要能够满足某个目标或受到某个条件的限制。根据限制条件的不同，从而得到不同的结果。

进而在设计算法时，要注意的是三维模型是离散的模型。虽然在光滑的曲面上，很多公式和原理可以互相推导，表达的是同一个现象。但在离散化时，可能某个公式更适合某个特殊的三维模型处理操作。而用另一个在光滑曲面上等价的推导，离散化后得到的效果并不好。因此，光滑曲面上的理论，定理可以知道离散算法的设计，但需要根据特定的问题来采用特定的离散化方法。

得到一个能量函数的最小值，就需要对此能量函数求导。假如能量函数是未知变量的二次形式，那么能量函数的求导得到的就是位置变量的线性方程组。因此在能量函数的设计中，目标是二次形式的能量函数。

一个顶点的一层邻域参数化映射图示如图 11-2 所示。

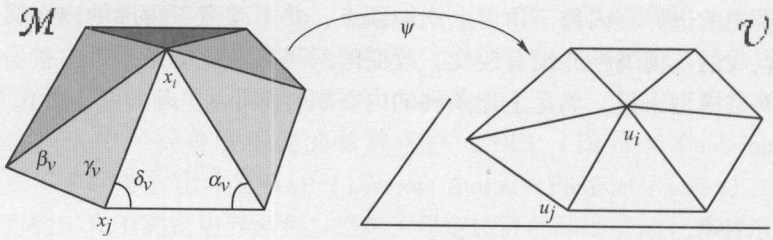


图 11-2　一层邻域映射

1. 狄利克雷能量函数

1）光滑曲面

狄利克雷能量（Dirichlet Energy）定义为：对于一个微分曲面到 X 到曲面 x 的光滑映射 U 的狄利克雷能量公式为此映射的梯度的 L_2。

范数：

$$E_D=\frac{1}{2}\int_X |\nabla u|^2 \mathrm{d}A \tag{11-6}$$

根据公式，此能量是正值，是对映射 u 所产生的扭曲的度量。如果映射 u 的映射域面积计算为：

$$A(u)=\int_X \det(u)\,\mathrm{d}A \tag{11-7}$$

那么可以推导出狄利克雷能量有个下限值：

$$E_D \geqslant A(u) \tag{11-8}$$

如果一个映射在把边界点的映射固定的前提下，使狄利克雷能量最小化，那么此映射称为和谐映射（Harmonic）。之所以称为和谐映射是因为映射 u 满足下面的条件。

$$\Delta u = 0 \tag{11-9}$$

如果 u 使狄利克雷能量得到最小的下限值，也就是映射域 $A(u)$，那么此映射称为保角映射。因此可以定义一个保角能量：

$$E_C(u) = E_D(u) - A(u) \tag{11-10}$$

如果此能量为零，那么就得到一个保角映射。根据黎曼定理，一个曲面总有一个保角映射，但如果把边界上点的映射事先固定的话，就无法获得保角映射。

2）离散三维模型

上述的映射都是相对于于光滑曲面，在离散的曲面，也就是三维网格的情况下，需要对映射做离散化处理。

（1）一个顶点的一层领域上的狄利克雷能量函数定义为：

$$E_A = \sum_{\text{orientededges}(i,j)} \cot\alpha_{ij} \mid u_i - u_j \mid^2 \tag{11-11}$$

（2）狄利克雷能量函数的定义需要知道参数化映射前的角度和参数化映射后的顶点位置，也就是边长。其中 $\mid u_i - u_j \mid$ 是映射后的边长，α_{ij} 是三维模型上映射前这条边的相对角度。

（3）一个顶点一层邻域的面积和狄利克雷能量函数有一定的关系。狄利克雷能量函数的最小值等于面积的值。

（4）面积的计算公式为：

$$\text{Area} = \frac{1}{2}\int_m |f_u + f_v| \, du dv \leqslant \frac{1}{2}\int_m |f_H| |f_v| \, du dv \tag{11-12}$$

$$\leqslant \frac{1}{4}\int_m (f_u^2 + f_v^2) \, du dv = \text{狄利克雷能量函数} \tag{11-13}$$

（5）假如 $f_u \cdot f_v = 0$，那么第一个不等号就变为等号，假如 $|f_u| = |f_v|$，那么第二个不等号就变为等号。因此狄利克雷能量的最小值和面积相等，假如映射是保角映射，那么狄利克雷能量获得最小值。

（6）对狄利克雷能量函数求导得到最小值为：

$$\frac{\partial E_A}{\partial u_i} = \sum_{j \in \lambda(i)} (\cot\alpha_{ij} + \cot\beta_{ij})(u_i - u_j) = 0 \tag{11-14}$$

（7）根据上述推导，得到一个线性系统，未知数是参数化后的顶点位置。求解这个线性系统就可以得到狄利克雷能量函数的最小值。

通过狄利克雷能量函数就把三维模型的面积内在属性容纳入了数学系统。上述方程也就是和谐参数化的方程。

2. Chi 能量函数

除了狄利克雷能量函数之外，还有 Chi 能量函数。狄利克雷能量函数的最小值保证角度在映射后不变，而 Chi 能量函数最小值保持面积在映射后不变。

$$E_\chi = \sum_{j \in N(i)} \frac{(\cot\gamma_{ij} + \cot\delta_{ij})}{|x_i - x_j|^2}(u_i - u_j)^2 \tag{11-15}$$

式中，$x_i - x_j$ | 是参数化映射前的边长；(u_i, u_j)是映射后顶点的位置。

对这个能量函数求导，也得到一个线性方程组。

$$\frac{\partial E_\chi}{\partial u_i} = \sum_{j \in N(i)} \frac{(\cot\gamma_{ij} + \cot\delta_{ij})}{|x_i - x_j|^2}(u_i - u_j) = 0 \tag{11-16}$$

从而总的能量函数为：

$$E = \lambda E_A + \mu E_\chi \tag{11-17}$$

式中，λ、μ 是两个实数常数。

11.3 线性系统构建

DCP 和 DAP 参数化算法和前几章介绍的算法一样，也需要构建一个线性系统。这个线性系统的构建是基于能量函数的求导得到的极值。线性系统根据有固定边界和没有固定边界需要分别进行构建。

假如预先固定了边界，在这个限制条件下，求得上述能量函数的最小值。根据狄利克雷能量函数和 Chi 能量函数的求导得到的一个线性系统，线性系统的形式和构建如下。

第一步：线性系统形式。

$$MU = \begin{bmatrix} \lambda M^A + \mu M^\chi & \\ 0 & I \end{bmatrix} \begin{bmatrix} U^{\text{internal}} \\ U^{\text{boundary}} \end{bmatrix} = \begin{bmatrix} 0 \\ C^{\text{boundary}} \end{bmatrix} = C \tag{11-18}$$

第二步：M 是线性系统中的矩阵，U 是待定的顶点位置，C^{boundary}是边界上的顶点位置。

第三步：矩阵 M 由两部分组成，其中的 M^A 元素计算如下。

$$M_{ij}^A = \begin{cases} \cot(\alpha_{ij}) + \cot(\beta_{ij}), & j \in N(i) \\ -\sum_{k \in N(i)} M_{ik}^A, & i = j \\ 0, & \text{其他} \end{cases} \tag{11-19}$$

第四步：矩阵 M 的另一个部分 M^χ 的元素计算如下。

$$M_{ij}^\chi = \begin{cases} (\cot(\gamma_{ij}) + \cot(\delta_{ij}))/|x_i - x_j|^2, & j \in N(i) \\ -\sum_{k \in N(i)} M_{ik}^\chi, & i = j \\ 0, & \text{其他} \end{cases} \tag{11-20}$$

第五步：DCP 参数化的结果是全局最优的，也就是保角的，而 DAP 参数化的结果是局部最优的，也就是只在顶点的 1 - 环邻域内保面积，而不是全局保面积。

第六步：狄利克雷能量函数相对于三角形某个顶点的导数等于这个顶点对面的边旋转 90°。即顶点变化如下。

$$(x, y) \rightarrow (-y, x)$$

第七步：上述过程需要预先确定边界上顶点的位置，但如果边界上的三角形满足：

$$\sum_{\Delta ijk} \cot\alpha(u_i - u_j) + \cot\beta(u_i - u_k) = \sum_{\Delta ijk} R^{90}(u_k - u_j) \tag{11-21}$$

那么就不需要预先确定边界上顶点的而为之，只需要确定两个顶点位置就可以求解整个方程组。

第八步：其中 R^{90}表示旋转 90°。

第九步：上述线性方程是 $n-2$ 阶（Rank Deficiency）的，故得到的退化（Degenerate）解是个常数。因此需要确定两个顶点的映射，此两个顶点的映射使所有顶点的旋转、缩放、位移唯一确定。

11.4　DCP 和 DAP 参数化核心代码

11.4.1　固定边界

整个算法的核心是构建线性系统的矩阵和向量。也就是两个能量函数求导的出来的矩阵，以及在自由边界的情况下，对边界上面的限制条件的设定。首先在固定边界的情况下，根据如下的线性系统构建两个矩阵。

$$MU=\begin{bmatrix}\lambda M^A+\mu M^K & \\ 0 & I\end{bmatrix}\begin{bmatrix}U^{\text{internal}}\\ U^{\text{boundary}}\end{bmatrix}=\begin{bmatrix}0\\ C^{\text{boundary}}\end{bmatrix}=C$$

（1）固定边界情况下，构建狄利克雷能量函数求导得出的矩阵 M^A。

```
private SparseMatrix BuildMatrixEA()
{
    int n = Mesh. Vertices. Count;
    SparseMatrix Ea = new SparseMatrix(n,n);
    foreach (TriMesh. Vertex item in Mesh. Vertices)
    {
      if (!item. OnBoundary)
      {
       foreach (TriMesh. HalfEdge hf in item. HalfEdges)
       {
         double alpha = TriMeshUtil. ComputeAngle(hf. Next. Opposite,
                                                  hf. Previous);
         double beta = TriMeshUtil. ComputeAngle(hf. Opposite. Next. Opposite,
                                                 hf. Opposite. Previous);
         Ea[item. Index,hf. ToVertex. Index] = 1 /Math. Tan(alpha)
                                             + 1/Math. Tan(beta);
       }
    }
}
int i = 0;
foreach (List < SparseMatrix. Element > row in Ea. Rows)
{
         double sum = 0;
         foreach (SparseMatrix. Element rowItem in row)
         {
```

```
                sum += rowItem.value;
            }
            Ea[i,i] = -sum;
            i++;
        }

        return Ea;
    }
```

（2）固定边界情况下，构建 Chi 能量函数求导得出的矩阵 M^x。

```
private SparseMatrix BuildMatrixEX()
{
  int n = Mesh.Vertices.Count;
  SparseMatrix Ex = new SparseMatrix(n,n);
  foreach (TriMesh.Vertex item in Mesh.Vertices)
  {
    if (!item.OnBoundary)
    {
     foreach (TriMesh.HalfEdge hf in item.HalfEdges)
     {
       double delta = TriMeshUtil.ComputeAngle(hf.Opposite,hf.Next);
       double ceta = TriMeshUtil.ComputeAngle(hf.Opposite,
                                              hf.Opposite.Previous.Opposite);
       Vector3D detaVector = item.Traits.Position -
                             hf.ToVertex.Traits.Position;
       Ex[item.Index,hf.ToVertex.Index] = (1/Math.Tan(ceta)
                                          +1/Math.Tan(delta))
                                          /detaVector.LengthSquared;
     }
  }
}
    int i=0;
    foreach (List<SparseMatrix.Element> row in Ex.Rows)
    {
        double sum=0;
        foreach (SparseMatrix.Element rowItem in row)
        {
            sum += rowItem.value;
        }
        Ex[i,i] = -sum;
        i++;
    }
```

```
        return Ex;
    }
```

(3) 根据权重设置两个矩阵之和。

```
SparseMatrix Ex = BuildMatrixEX( );
SparseMatrix Ea = BuildMatrixEA( );
Ex. Multiply( weightDAP) ;
Ea. Multiply( weightDCP) ;
SparseMatrix LeftMatrix = Ex. Add( Ea) ;
```

(4) 为矩阵增加固定边界的限制条件。

```
public void AddBoundary( SparseMatrix m)
{
    foreach ( TriMesh. Vertex boundaryV in Mesh. Vertices)
    {
        if ( boundaryV. OnBoundary)
        {
            m[ boundaryV. Index, boundaryV. Index] = 1;
        }
    }
}
```

(5) 从而整个线性系统的矩阵构建如下。

```
private SparseMatrix BuildMatrixLeft( double weightDCP, double weightDAP)
{
    SparseMatrix Ex = BuildMatrixEX( );
    SparseMatrix Ea = BuildMatrixEA( );
    Ex. Multiply( weightDAP) ;
    Ea. Multiply( weightDCP) ;
    SparseMatrix LeftMatrix = Ex. Add( Ea) ;
    AddBoundary( LeftMatrix) ;
    return LeftMatrix;
}
```

(6) 根据边界顶点设定的位置，构建线性系统的右边向量。

```
private double[ ][ ] BuildRightB( ParaBoundary boundary)
{
    int m = Mesh. Vertices. Count;
    int n = 2;
    double[ ][ ] RightB = new double[ n][ ];
    for ( int j = 0; j < n; j ++ )
    {
```

```
            RightB[j] = new double[m];
        }
        for (int i = 0;i < m;i ++)
        {
            RightB[0][i] = 0;
            RightB[1][i] = 0;
        }
        for (int i = 0;i < boundary.UV.Boundary.Count;i ++)
        {
            TriMesh.Vertex vertex = boundary.UV.Boundary[i];
            RightB[0][vertex.Index] = boundary.UV.Pos[i].x;
            RightB[1][vertex.Index] = boundary.UV.Pos[i].y;
        }
        return RightB;
    }
```

（7）求解线性系统，得到位置顶点的位置。

```
    public void Parameterlize(double weight1,double weight2)
    {
        ParaBoundary boundary = new ParaBoundary(Mesh);
        boundary.SetUpBoundary();
        SparseMatrix LeftMatrix = BuildMatrixLeft(weight1,weight2);
        double[][] RightB = BuildRightB(boundary);
        SolveLinearSystem(LeftMatrix,RightB);
    }
```

11.4.2 自由边界

自由边界和固定边界在构建矩阵时有很大不同。固定边界可以把两个未知变量 u、v 分别求解。而自由边界 u、v 由于边界的设定，而无法分别求解。因此，自由边界的矩阵构建需要把 u、v 混到到一起。也就是对于顶点数为 n 的三维模型，固定边界构建的能量函数求导矩阵大小为 $n \times n$ 的，而自由边界的矩阵大小为 $2n \times 2n$ 的。

（1）在自由边界情况下，构建狄利克雷能量函数求导得出的矩阵 M^A。

```
    public SparseMatrix BuildMatrixEX(TriMesh mesh)
    {
        int n = mesh.Vertices.Count;
        SparseMatrix Ex = new SparseMatrix(2 * n,2 * n);
        foreach (TriMesh.Vertex item in mesh.Vertices)
        {
         if (!item.OnBoundary)
         {
```

```
        foreach (TriMesh.HalfEdge hf in item.HalfEdges)
        {
        double delta = TriMeshUtil.ComputeAngle(hf.Opposite,hf.Next);
        double ceta = TriMeshUtil.ComputeAngle(hf.Opposite,
                                                hf.Opposite.Previous.Opposite);
        Vector3D detaVector = item.Traits.Position
                                - hf.ToVertex.Traits.Position;

        int row = 2 * item.Index;
        int col = 2 * hf.ToVertex.Index;
        Ex[row,col] = (1/Math.Tan(ceta) + 1/Math.Tan(delta))
                        / detaVector.LengthSquared;
        Ex[row + 1,col + 1] = (1/Math.Tan(ceta) + 1/Math.Tan(delta))
                                / detaVector.LengthSquared;
        }
    }
    }
    int i = 0;
    foreach (List<SparseMatrix.Element> row in Ex.Rows)
    {
        double sum = 0;
        foreach (SparseMatrix.Element rowItem in row)
        {
            sum += rowItem.value;
        }
        Ex[i,i] = -sum;
        i++;
    }
    return Ex;
}
```

(2) 在自由边界情况下，构建 Chi 能量函数求导得出的矩阵 M^x。

```
public SparseMatrix BuildMatrixEA(TriMesh mesh)
{
    int n = mesh.Vertices.Count;
    SparseMatrix Ea = new SparseMatrix(2 * n,2 * n);
    foreach (TriMesh.Vertex item in mesh.Vertices)
    {
        if (!item.OnBoundary)
        {
        foreach (TriMesh.HalfEdge hf in item.HalfEdges)
        {
```

```
            double alpha = TriMeshUtil. ComputeAngle( hf. Next. Opposite,
                                                    hf. Previous) ;
            double beta = TriMeshUtil. ComputeAngle( hf. Opposite. Next. Opposite,
                                                    hf. Opposite. Previous) ;
            int row = 2 * item. Index;
            int col = 2 * hf. ToVertex. Index;
            Ea[ row, col] = 1/Math. Tan( alpha) + 1/Math. Tan( beta) ;
            Ea[ row + 1, col + 1] = 1/Math. Tan( alpha) + 1/Math. Tan( beta) ;
          }
      }
    }
    int i = 0;
    foreach ( List < SparseMatrix. Element > row in Ea. Rows)
    {
        double sum = 0;
        foreach ( SparseMatrix. Element rowItem in row)
          {
              sum += rowItem. value;
          }
          Ea[ i, i] = - sum;
          i ++ ;
    }
    return Ea;
}
```

（3）根据公式 $\sum_{\Delta ijk} \cot\alpha(u_i - u_j) + \cot\beta(u_i - u_k) = \sum_{\Delta ijk} R^{90}(u_k - u_j)$，增加自由边界的限制条件。

```
private void AddBoundaryCondition( SparseMatrix Ea, SparseMatrix Ex)
{
 List < List < TriMesh. Vertex >> bounds =
                        TriMeshUtil. RetrieveBoundaryAllVertex( Mesh) ;
 foreach ( List < TriMesh. Vertex > vb in bounds)
 {
   foreach ( TriMesh. Vertex Vi in vb)
   {
   foreach ( TriMesh. HalfEdge hf in Vi. HalfEdges)
   {
        if ( hf. OnBoundary)
        {
            continue;
        }
```

```
            TriMesh. HalfEdge Hij = hf;
            TriMesh. HalfEdge Hik = hf. Previous. Opposite;
            TriMesh. Vertex Vj = Hij. ToVertex;
            TriMesh. Vertex Vk = Hik. ToVertex;

            double alpha = TriMeshUtil. ComputeAngle( hf. Next. Opposite,
                                                      hf. Previous) ;

            double beta = TriMeshUtil. ComputeAngle( hf. Opposite, hf. Next) ;
            double cotAlpha = 1/Math. Tan( alpha) ;
            double cotBeta = 1/Math. Tan( beta) ;

            Ea[ 2 * Vi. Index, 2 * Vi. Index] += ( cotAlpha + cotBeta) ;
            Ea[ 2 * Vi. Index, 2 * Vj. Index] += - cotAlpha;
            Ea[ 2 * Vi. Index, 2 * Vk. Index] += - cotBeta;

            Ea[ 2 * Vi. Index, 2 * Vk. Index + 1] += 1;
            Ea[ 2 * Vi. Index, 2 * Vj. Index + 1] += -1;

            Ea[ 2 * Vi. Index + 1, 2 * Vi. Index + 1] += ( cotAlpha + cotBeta) ;
            Ea[ 2 * Vi. Index + 1, 2 * Vj. Index + 1] += - cotAlpha;
            Ea[ 2 * Vi. Index + 1, 2 * Vk. Index + 1] += - cotBeta;

            Ea[ 2 * Vi. Index + 1, 2 * Vk. Index] += -1;
            Ea[ 2 * Vi. Index + 1, 2 * Vj. Index] += 1;
        }
    }
}


Ea. AddRow( ) ;
Ex. AddRow( ) ;
Ea[ Ea. Rows. Count - 1, 2 * boundary. fixVertexOne. Index] = 1;
Ex[ Ea. Rows. Count - 1, 2 * boundary. fixVertexOne. Index] = 1;

Ea. AddRow( ) ;
Ex. AddRow( ) ;
Ea[ Ea. Rows. Count - 1, 2 * boundary. fixVertexOne. Index + 1] = 1;
Ex[ Ea. Rows. Count - 1, 2 * boundary. fixVertexOne. Index + 1] = 1;

Ea. AddRow( ) ;
Ex. AddRow( ) ;
Ea[ Ea. Rows. Count - 1, 2 * boundary. fixVertexTwo. Index] = 1;
```

```
        Ex[Ea.Rows.Count-1,2*boundary.fixVertexTwo.Index]=1;

        Ea.AddRow();
        Ex.AddRow();
        Ea[Ea.Rows.Count-1,2*boundary.fixVertexTwo.Index+1]=1;
        Ex[Ea.Rows.Count-1,2*boundary.fixVertexTwo.Index+1]=1;
    }
```

（4）构建线性系统的左边矩阵。

```
    private SparseMatrix BuildMatrixLeft(double weightDCP,double weightDAP)
    {
        SparseMatrix Ex=BuildMatrixEX(Mesh);
        SparseMatrix Ea=BuildMatrixEA(Mesh);
        Ex.Multiply(weightDAP);
        Ea.Multiply(weightDCP);
        AddBoundaryCondition(Ea,Ex);
        SparseMatrix LeftMatrix=Ex.Add(Ea);
        return LeftMatrix;
    }
```

（5）根据固定的两个顶点位置，构建线性系统的右边向量。

```
    private double[] BuildRightB()
    {
        int m=Mesh.Vertices.Count;
        double[] RightB=new double[2*m+4];
        for(int i=0;i<2*m;i++)
        {
            RightB[i]=0;
        }
        RightB[2*m]=    boundary.handleFirst.x;
        RightB[2*m+1]=boundary.handleFirst.y;
        RightB[2*m+2]=boundary.handleSecond.x;
        RightB[2*m+3]=boundary.handleSecond.y;
        return RightB;
    }
```

（6）对于两个位置向量 u、v 联合求解。

```
    public void Parameterlize(double weight1,double weight2)
    {
        boundary=new ParaBoundary(Mesh);
        boundary.GetFixTwoPoint();
        SparseMatrix LeftMatrix=BuildMatrixLeft(weight1,weight2);
```

```
        double[ ] RightB = BuildRightB( );
        double[ ] u = LinearSystem. Instance. SloveLeastSquare( ref LeftMatrix,
                                                                ref RightB);
        Convert( u);
        FreeSystem( ref LeftMatrix, ref RightB);
    }
```

11.5　DCP 和 DAP 实验效果分析

DCP、DAP 参数化算法从保持三维模型的内在属性出发，构建能量函数，从而得到一个保持角度，或者保持面积的二维平面。DCP、DAP 参数化既可以是固定边界，又可以是自由边界。对于自由边界的 DCP 来说，效果和 LSCM 相等。虽然推导的出发点和过程不一样，但最终得到的是同一个数学线性系统。可以定义两个权重把 DCP 和 DAP 混合起来，也就是利用两个权重在保持角度不变和保持面积不变之间进行平衡。这种参数化算法从保持三维模型的内在属性出发设计，因此可以比较好地保持三维模型的许多内在属性。例如，两个三维模型形状一样，但是顶点数不同，那么得到的二维平面是一样的。

1. DCP 实验效果

DCP 是保角参数化，目标是尽可能地保持角度不变，在固定边界的情况下，效果如图 11-3 所示。图中的兔子半身三维模型经过 DCP 算法后变为一个具有圆形边界的平面，在贴图之后可以看出在兔子耳朵地方扭曲比较大。

DCP 在自由边界下和 LSCM 的结果一样，如图 11-4 所示。从各种不同形状的三维模型两种参数化结果可以看出，DCP 和 LSCM 算法参数化效果是一致的，得到几乎一样的平展结果。

2. DAP 实验效果

DAP 参数化目标是保持面积不变。和 DCP 相比，DAP 参数化的结果得到更小的面积扭曲，但有更大的角度扭曲，面具模型的两种参数化算法的结果扭曲对量对比如图 11-5 所示。

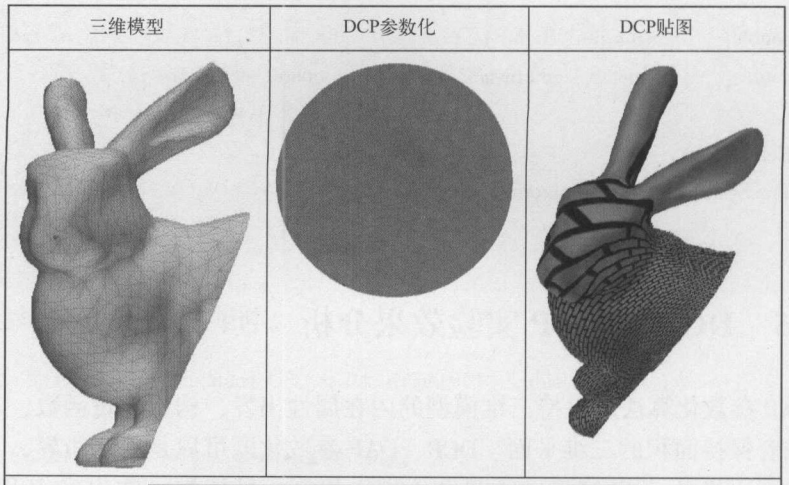
DCP 和 DAP 还可以作用于多边界的三维模型，多边界的三维模型 DCP 和 DAP 效果对比如图 11-6 所示，从中可以看出，DCP 可以更好地维持原来边界的形状。

3. 不同分辨率的三维模型 DCP、DAP 效果

DCP 和 DAP 设计的起点是保持三维模型的内在属性不变，因此在不同分辨率的情况下，三维模型的参数化结果差别不大。如图 11-7 所示为不同分辨率的兔子模型在参数化后贴图的结果几乎一样。

4. 不同固定顶点的三维模型 DCP 效果

LSCM、DCP 等自由边界的参数化算法需要确定两个顶点的映射。因此参数化的结果依赖于这两个顶点的选择。如图 11-8 所示，红色和蓝色的点是选择的两个固定顶点。图 11-8 中三列分别表示选择不同的固定顶点。顶点不同得到的参数化结果不同，但假如两个顶点距离最远，得到的参数化结果就比较好。第一行表示兔子的三维模型上选择两个不同的固定顶点。第二行表示兔子在两个固定顶点下相应的参数化结果。第三行是三个半球三维模型不同的固定顶点，第四行是相应的参数化结果。从中可以看出参数化结果随着选择的固定顶点的不同而不同。



Index	Distortion Name	Distortion Value
0	Name	DCPDAP
1	Angle Avg Distortion	0.256439206004321
2	Angle Total Distortion	2181.27188627276
3	Angle Max Distortion	1.89656300576043
4	Area Total Distortion	1.05659741705308
5	Area Max Distortion	0.00367286569306019
6	Edge Total Distortion	0.80640684130934
7	Edge Max Distortion	0.00103287272359722
8	L2 Distortion	45.327862177708
9	L8 Distortion	533.489558965664
10	QuasiConformal	1.99688848506474
11	DegenerateTriangle	0

图 11-3　DCP 参数化效果

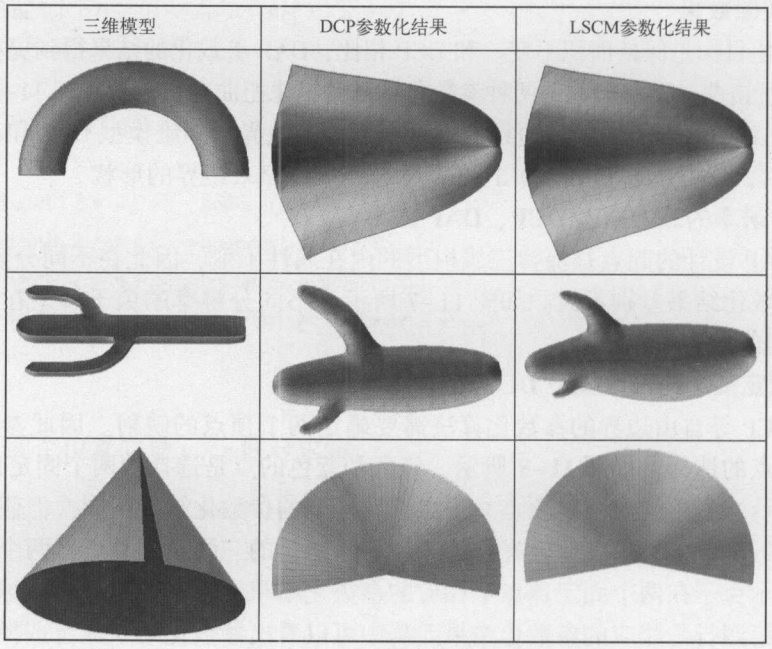
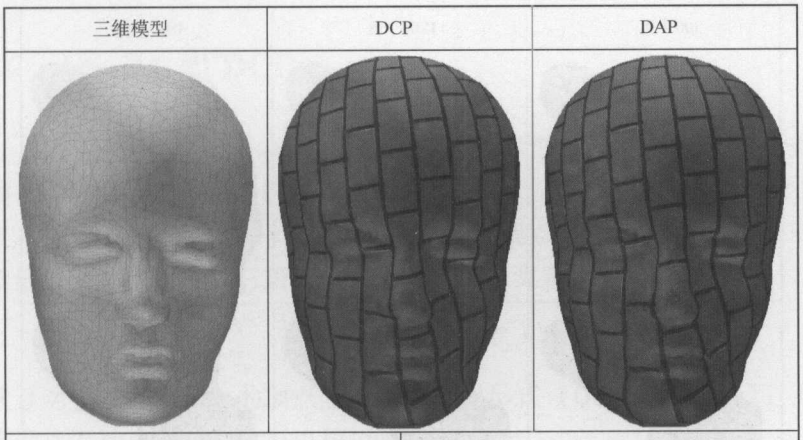


图 11-4　DCP 和 LSCM 参数化对比



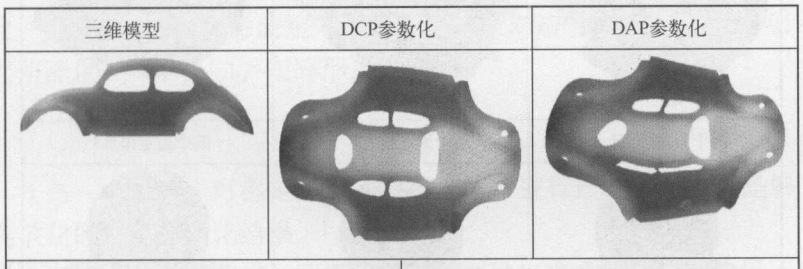
DCP度量

Index	Distortion Name	Distortion Value
0	Name	DCPDAP
1	Angle Avg Distortion	0.148304298877386
2	Angle Total Distortion	1031.30809439334
3	Angle Max Distortion	0.850116851367715
4	Area Total Distortion	0.253452413034052
5	Area Max Distortion	0.00139779958086692
6	Edge Total Distortion	0.168270380671593
7	Edge Max Distortion	0.000475756594934277
8	L2 Distortion	1.09524813867128
9	L8 Distortion	3.0155665260348
10	QuasiConformal	1.35813542350261
11	DegenerateTriangle	0

DAP度量

Index	Distortion Name	Distortion Value
0	Name	DCPDAP
1	Angle Avg Distortion	0.161597884191451
2	Angle Total Distortion	1123.75168666735
3	Angle Max Distortion	0.954520782151938
4	Area Total Distortion	0.22177522238086
5	Area Max Distortion	0.00130968155009048
6	Edge Total Distortion	0.152780260630349
7	Edge Max Distortion	0.000480728351630368
8	L2 Distortion	1.07899785741169
9	L8 Distortion	4.54528112281508
10	QuasiConformal	1.38699969116984
11	DegenerateTriangle	0

图 11-5　DAP 参数化效果



DCP扭曲度量

Index	Distortion Name	Distortion Value
0	Name	DCPFreeBoundary
1	Angle Avg Distortion	0.0128655295767623
2	Angle Total Distortion	70.8118747904996
3	Angle Max Distortion	0.316182883828637
4	Area Total Distortion	0.223869250836401
5	Area Max Distortion	0.000302818626851387
6	Edge Total Distortion	0.117233477975479
7	Edge Max Distortion	0.000113345784879081
8	L2 Distortion	1.04186612251316
9	L8 Distortion	1.467488269133 83
10	QuasiConformal	1.02203021371466
11	DegenerateTriangle	0

DAP扭曲度量

Index	Distortion Name	Distortion Value
0	Name	DCPFreeBoundary
1	Angle Avg Distortion	0.0236725694608848
2	Angle Total Distortion	130.29382231271
3	Angle Max Distortion	0.725995705260248
4	Area Total Distortion	0.296254732752613
5	Area Max Distortion	0.000745863739411084
6	Edge Total Distortion	0.151253765373001
7	Edge Max Distortion	0.000267147169721457
8	L2 Distortion	1.07032635723353
9	L8 Distortion	5.1151285818318
10	QuasiConformal	1.04211033479082
11	DegenerateTriangle	0

图 11-6　多边界三维模型参数化效果

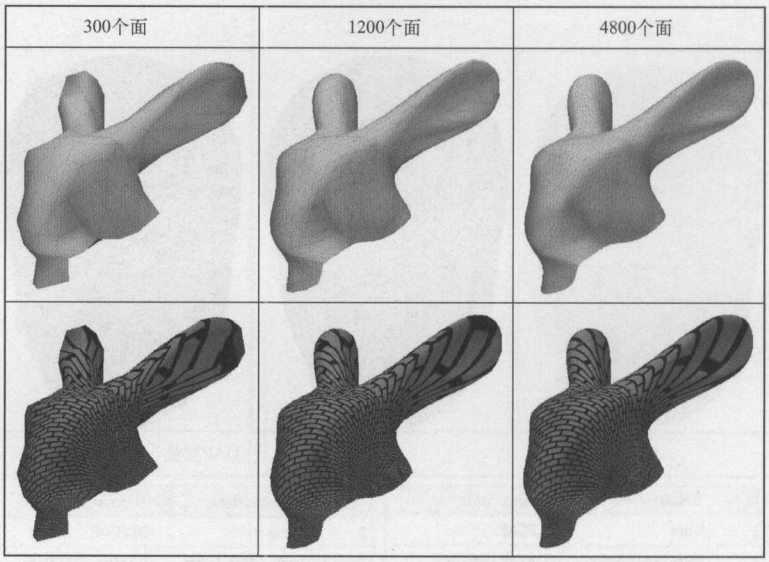


图 11-7　不同分辨率的三维模型 DCP 参数化效果

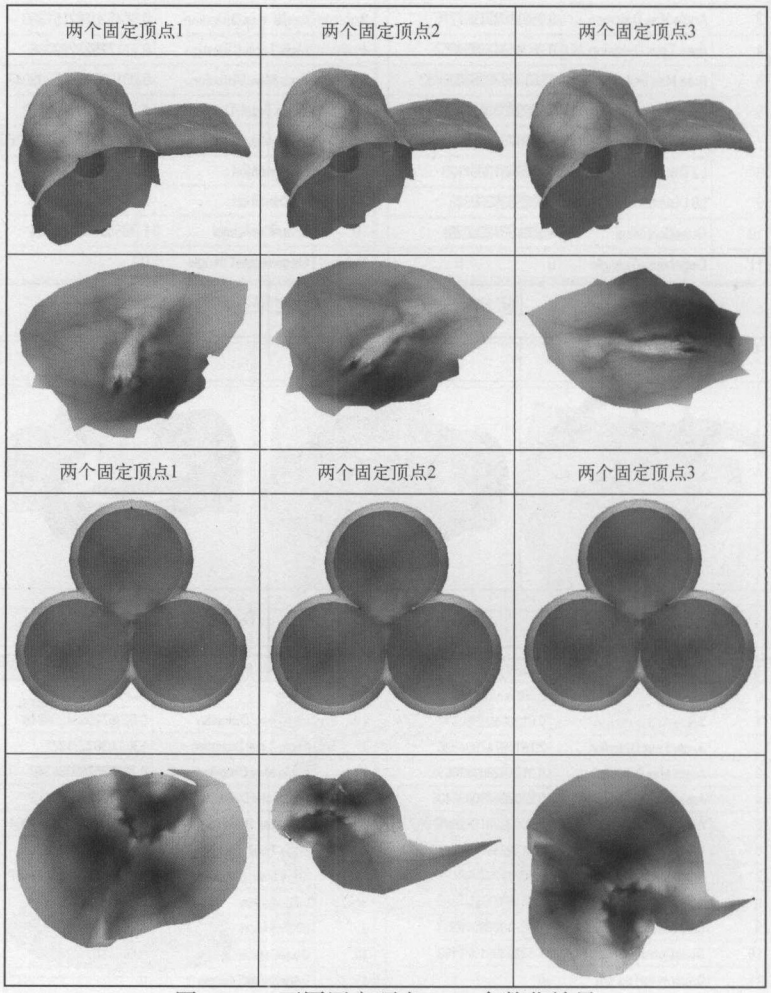


图 11-8　不同固定顶点 DCP 参数化效果

11.6　LSCM 和 DCP

DCP（Discrete Conformal Parameterization）指的是离散保角参数化。和 LSCM 算法的构造和推导过程不一样，但推导出来的结果公式是一致的。也就是非固定边界的 DCP 和 LSCM 的推导出发点不一样，但结果是一样的。DCP 方法的视点是狄利克雷能量。对于三维网格上的每个三角形映射到平面上的三角形，离散化的狄利克雷能量可以表示为：

$$E_D(u) = \sum_{e_{ij}} \frac{1}{4}(\cot(\theta_{ij}) + \cot(\theta_{ji}))\ (u_i - u_j)^2 \tag{11-22}$$

这个公式是对于二维平面上的顶点坐标 u_i 的二次形式（Quadratic Form），其中相对应的系数是由三维网格上计算出来的。可以用矩阵表示为：

$$E_D(u) = \frac{1}{2}u^t L_D u \tag{11-23}$$

其中 L_D 是 $2V \times 2V$ 的稀疏、对称矩阵。L_D 也被称为离散拉普拉斯矩阵。参数化域的上单个三角形面积可以计算如下：

$$A_T(u) = \sum_{e_{ij} \in T} \frac{1}{2}(u_i v_j - u_j v_i) \tag{11-24}$$

所有面积之和可以只使用参数化域边界上的边进行计算：

$$A(u) = \sum_{e_{ij} \in \partial u} \frac{1}{2}(u_i v_j - u_j v_i) \tag{11-25}$$

用矩阵形式表示为：

$$A(u) = \frac{1}{2}u^t A u \tag{11-26}$$

那么 A 是一个 $2V \times 2V$ 极度稀疏矩阵，矩阵里只有对应着边界上的点的项才不为零。因此离散化的保角能量 $E_C = E_D - A$，矩阵形式为：

$$E_C(u) = \frac{1}{2}u^t L_C u \tag{11-27}$$

其中 $L_C = L_D - A$ 是一个稀疏、对称矩阵。因此对 $E_C(u)$ 进行最小化，也就是解一个线性方程就可以得到离散化的保角参数化结果。

上述结果也可以用另一种方式，即 LSCM 算法推导出来。假如参数化后 u 和 v 坐标的梯度满足在每个三角形中尽可能的正则相交，也就是下面的能量被最小化：$\perp$ 表示逆时针旋转 90°。

$$E_{\mathrm{LSCM}}(u) = \int_\chi \frac{1}{2} \mid \nabla u^\perp - \nabla v \mid^2 \mathrm{d}A \tag{11-28}$$

但上述公式可以简化为：

$$\begin{aligned} E_{\mathrm{LSCM}}(u) &= \int_\chi \frac{1}{2}(\nabla u^\perp \cdot \nabla u^\perp + \nabla v \cdot \nabla v - 2\nabla u^\perp \cdot \nabla v)\mathrm{d}A \\ &= \int_\chi \frac{1}{2}(\nabla u \cdot \nabla u + \nabla v \cdot \nabla v - 2\nabla u \times \nabla v)\mathrm{d}A \\ &= E_D(u) - A(u) \end{aligned} \tag{11-29}$$

从而得到和 DCP 相同的结果。

不固定边界的 DCP 和 LSCM 推导的能量函数 $E_C(u)=\frac{1}{2}u^tL_Cu$ 最小化后，得到如下线性方程组：

$$L_Cu=0 \tag{11-30}$$

从而两个算法得到的最终线性系统是一致的。

11.7 自由边界参数化

LSCM 和 DCP 都是自由边界参数化，但两种算法的设计、构造、推导起点不同，最终得到的数学模型的形式一样。自由边界参数化算法还可以通过如下方式进行推导。

第一步：对于一个三角形，用 r_i 表示边长 $\{x_0,x_i\}$，用 R^{90} 表示 90°的旋转 $(u,v)\rightarrow(-v,u)$，那么在图 11-9 中

$$R^{90}(y_2-y_1)=R^{90}(y_2-x_0)-R^{90}(y_1-x_0) \tag{11-31}$$

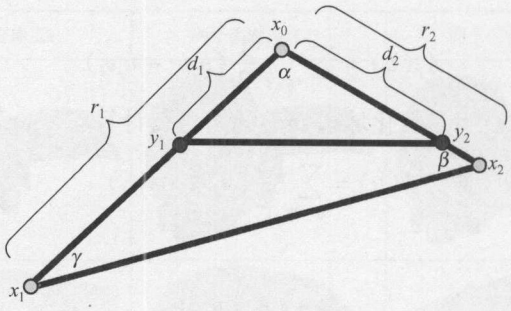


图 11-9　三角形图示

第二步：因为

$$R^{90}(y_1-x_0)=\frac{d_1}{\sin(\alpha)}\cdot\frac{x_2-x_0}{\gamma_2}-d_1\cot(\alpha)\cdot\frac{x_1-x_0}{\gamma_1} \tag{11-32}$$

$$R^{90}(y_2-x_0)=d_2\cot(\alpha)\cdot\frac{x_2-x_0}{\gamma_2}-\frac{d_2}{\sin(\alpha)}\cdot\frac{x_1-x_0}{\gamma_1} \tag{11-33}$$

第三步：那么可以得出：

$$R^{90}(y_2-y_1)=\left(d_2\cot(\alpha)-\frac{d_1}{\sin(\alpha)}\right)\cdot\frac{x_2-x_0}{\gamma_2}+\left(d_1\cot(\alpha)-\frac{d_2}{\sin(\alpha)}\right)\cdot\frac{x_1-x_0}{\gamma_1} \tag{11-34}$$

第四步：根据定义：

$$R^{90}(y_2-y_1)=R^{90}\left(d_2\frac{x_2-x_0}{\gamma_2}-d_1\frac{x_1-x_0}{\gamma_1}\right) \tag{11-35}$$

第五步：从而最终得到如下公式：

$$\left(d_2\cot(\alpha)-\frac{d_1}{\sin(\alpha)}\right)\cdot\frac{x_2-x_0}{\gamma_2}+\left(d_1\cot-\frac{d_2}{\sin(\alpha)}\right)\cdot\frac{x_1-x_0}{\gamma_1}=R^{90}\left(d_2\frac{x_2-x_0}{\gamma_2}-d_1\frac{x_1-x_0}{\gamma_1}\right) \tag{11-36}$$

第六步：假如 $(d_i=r_i)$，那么根据

$$\frac{r_2}{r_3}=\frac{\sin(\beta)}{\sin(\gamma)}$$

$$\sin(\gamma)=\sin(\alpha+\beta)$$

第七步：上述公式简化为：

$$\cot(\gamma)\cdot(x_2-x_0)+\cot(\beta)\cdot(x_1-x_0)=R^{90}(x_2-x_1) \tag{11-37}$$

第八步：从而得到 DCP 的公式。

第九步：假如($d_i=1$)，那么

$$-\cot(\alpha)+1/\sin(\alpha)=\tan(\alpha/2)$$

第十步：可以得出：

$$\frac{\tan(\alpha/2)}{\gamma_2}(x_2-x_0)+\frac{\tan(\alpha/2)}{\gamma_1}(x_1-x_0)=R^{90}\left(\frac{x_2-x_0}{\gamma_2}-\frac{x_1-x_0}{\gamma_1}\right)$$

也就是中值权重。

第十一步：为了把一个顶点表示为周边顶点的关系，把一个顶点所有相邻的三角形用上述公求和，那么可以得出公式：

$$\sum_{i=1}^{k}w_i(x_i-x_0)=0$$

$$w_i=\lambda_i/\sum_{i=1}^{k}\lambda_i \tag{11-38}$$

$$\lambda_i=\frac{1}{\gamma_i}\left(\frac{d_{i-1}}{\sin(\alpha_{i-1})}-d_i\cot(\alpha_i)\right)$$

第十二步：其中 α_i 是第 i 和第 $i+1$ 条边的夹角。β_i 和 α_i 是与 y_i 和 y_{i+1} 相对的角度。

第十三步：假如通过点 y_i 绘制一个和相应的边垂直的直线，那么这些直线相交形成一个多边形，多边形的边长为 l_i，那么可以得出如下公式，如图 11-10 所示。

$$\lambda_i=\frac{l_i}{r_i} \tag{11-39}$$

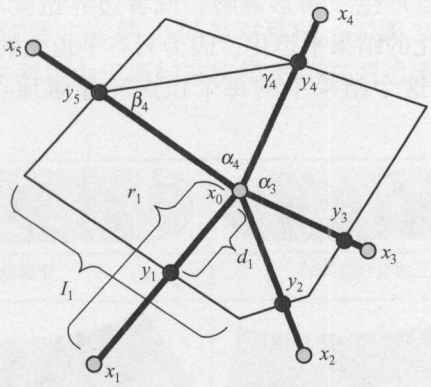


图 11-10　推导图示

第十四步：那么常见的三种权重是：

$$d_i=r_i,\quad w_i=\cot(\beta)+\cot(\gamma_{i-1})$$

$$d_i=1,\quad w_i=(\tan(\alpha_{i-1}/2)+\tan(\alpha_i/2))/\gamma_i \tag{11-40}$$

$$d_i=1/r_i,\quad w_i=(\cot(\beta_{i-1})+\cot(\gamma_i))/\gamma_i$$

第十五步：假如$\beta_{i-1}+\gamma_i<\pi$，那么权重w_i是正值。因为三角形$\{y_{i-1},x_0,y_i\}$是等腰三角形，所以中值权重总是正值。

第十六步：三角形内部的每个顶点可以得出：

$$\sum_{i=1}^{k} w_i(x_i-x_0)=R^{90}\left(d_2\frac{x_2-x_0}{\gamma_2}-d_1\frac{x_1-x_0}{\gamma_1}\right) \tag{11-41}$$

第十七步：因此内部的顶点落在邻居顶点的重心，但边界上的顶点不在重心，而是重心的偏移位置。虽然内部顶点可以把每个顶点的两个坐标u、v分开来表示，但在边界顶点两个坐标无法分开，所以整个线性系统是u、v混合在一起。不能分开来求解。这是一个$2V\times 2V$的线性系统。

第十八步：对于可扩展三维模型来说，得到的$2V\times 2V$的线性系统的秩是$2V-4$，这是因为这个线性方程组经过两个方向平移，一个旋转和一个缩放是不变的，因此有4个自由度。因此需要去掉4个自由度，得到唯一的解。可以通过确定两个顶点的位置，也就是两个顶点的u、v坐标，一共4个未知数的值来去掉自由度。

第十九步：对于不可扩展的三维模型来说，得到的$2V\times 2V$的线性系统的秩是$2V-2$，即只有一个唯一的退化解，所有顶点都是零，这个解只有两个方向的平移自由度。因此也需要固定两个边界顶点来避免退化解。同时在线性系统里去掉这两个顶点对应的边界的限制条件。

第二十步：对于三角形内的两个边，如果把一个边旋转相应角度，并缩放相应比例，那么两条边就重合。也就是：

$$\frac{\gamma_1}{\gamma_2}(x_2-x_0)=R^{\alpha}(x_1-x_0)$$

第二十一步：对于所有三角形，这构成一个线性方程组。这是一个$2T\times 2V$的线性系统，也可以和之前推导一样，对每个顶点邻居三角形用上述方程求和，得到一个$2V\times 2V$的线性系统。

自由边界参数化算法都会产生三角形翻转，或者边界扭度不是2π等非平坦化的结果。可以用迭代的方法把非平坦化的结果平坦化。因为只有平坦化的结果满足中值方程，也就是假如结果是平坦化的，那么这个结果中的每个顶点和邻居顶点的关系可以用中值权重来表示。

第 12 章 频谱参数化算法

12.1 算法特点

前几章讲述的各种参数化算法由于受到边界约束条件的限制，会造成很大扭曲。即使是基于自由边界的算法，如 LSCM、DCP 算法需要固定至少两个顶点，因此这两个顶点的限制条件造成了不必要的扭曲。频谱参数化算法不需要固定任何顶点，从而能够得到效果更好的结果。频谱参数化算法利用的是矩阵的特征值和特征向量，因此称为频谱参数化算法。频谱参数化算法（Spectral Conformal Parameterization）能够自动、有效地对三角形网格模型进行不固定边界的保角参数化。LSCM、DCP 等自由边界参数化算法由于限制位置而产生明显的人工缺陷，而频谱参数化的结果没有这样的缺陷。虽然频谱算法时间复杂度上需要求矩阵的特征向量，但相对于其他线性方法，质量上和数量上都有很大提高。频谱参数化方法使用菲德尔特征向量作为参数化的解，从而避免了需要事先确定两个顶点的映射。频谱参数化算法也是一种线性代数（Linear Algebra）的方法，和 LSCM/DCP 等其他线性方法一样，这种方法不能保证没有三角形翻转。频谱参数化算法的结果也没有不规则采样所造成的偏移（Bias）。也就是假如三维模型左右两边的采样点密度不一样，但模型是左右对称的，用 LSCM/DCP 算法参数化的结果丧失了对称性，但用频谱算法参数化的结果成功显示了原来三维模型的对称性。

如图 12-1（a）所示，三维模型左右两边是对称的。在右半边被细分后如图 12-1（b）所示，用频谱参数化得到的结果左右两边仍是对称的，如图 12-1（d）所示，但用 LSCM 参数化得到的展平结果的左右两边由于原始三维模型左右两边密度的不一致而变得不对称，如图 12-1（f）所示。

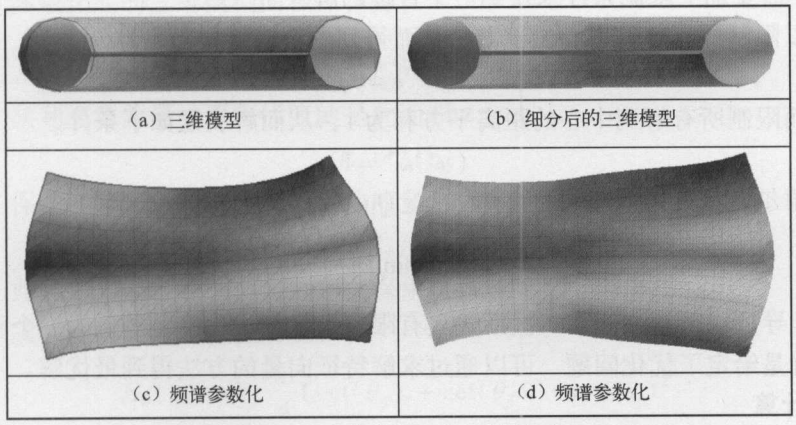
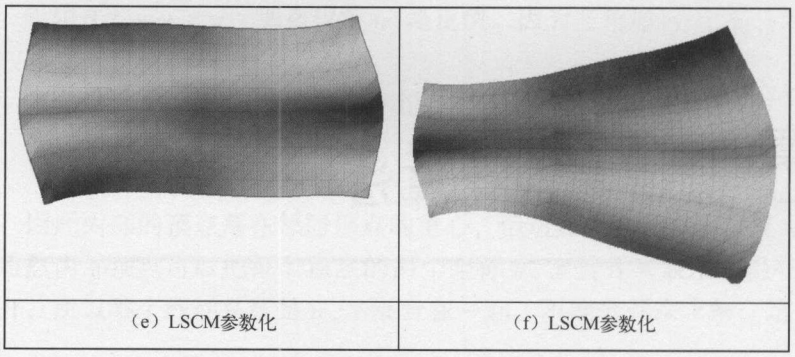


图 12-1　频谱参数化算法效果对比



（e）LSCM参数化　　（f）LSCM参数化

图 12-1　频谱参数化算法效果对比（续）

12.2　菲德勒向量

1. 概念和特点

在频谱参数化算法中，用到的最重要的数学工具是菲德尔向量，菲德尔向量是矩阵的一个特殊的特征向量，这个向量具有某些特点，正好能够满足参数化的要求。

（1）**定义**：对于一个对称半正定正值拉普拉斯矩阵 L 来说，最小的非零特征值对应的特征向量，称为菲德勒向量（Fiedler Vector）。

$$Lu^* = \lambda u^* \tag{12-1}$$

（2）菲德勒向量不仅是矩阵的最小非零特征向量，而且还是和矩阵相关的优化问题的最小值。假如矩阵的秩是 $n-k$，那么菲德勒向量也是如下二次形式线性约束的优化问题的最小值。

$$\begin{aligned} u^* &= \underset{u}{\operatorname{argmin}} u'Lu \\ u'e &= 0 \\ u'u &= 1 \end{aligned} \tag{12-2}$$

（3）其中 e 是 $n \times k$ 矩阵，由矩阵 L 的核（Kernel）构成。

（4）假如上述优化问题没有限制约束条件，那么优化问题的最小值就是 0。这是一个退化解。因此，需要加上限制条件来得到一个合理的解。

（5）可以限制所有解的中心位于零，从而得到的解满足如下条件。

$$(u^{*t}e = 0)$$

（6）进而限制所有解到中心的距离平方和为 1，从而解满足如下条件。

$$(u^{*t}u^* = 1)$$

（7）菲德尔向量是如下瑞利商（Rayleigh Quotient）函数的最小值。

$$u^* = \underset{u}{\operatorname{argmin}} \left| \frac{u^t L_C u}{u^t u} \right| \tag{12-3}$$

从上述推导可以得知，一个二次形式具有限制条件的优化问题等价为一个相应的特征向量问题。也就是给定了优化问题，可以通过求解特征向量的方法得到最优解。

2. 数值计算

为了使求解特征向量的数值计算更稳定，可对上述特征向量方程进行置换。

第一步：进行左右调换，从而使最小的特征向量变为求解最大的特征向量，也就是：

$$\mu = 1/\lambda \tag{12-4}$$

通常求最大的特征向量比求最小的特征向量稳定。

第二步：然后对 L_C 矩阵加上一个 εID 矩阵，其中 $\varepsilon = 10^{-8}$，来避免由于数值误差造成的拉普拉斯矩阵 L_C 为非正值半正定矩阵。

第三步：通过把特征方程进行下面的转换，可以在计算中移除多个特征向量的计算。

$$\left[B - \frac{1}{V_b} e_b e_b^t\right] u = \mu L_C u \tag{12-5}$$

第四步：其中 $e_b = Be$，也就是把 e 里面对应于内部点的值都设置为零。V_b 是边界点的数量。

第五步：这个特征方程的特征向量和之前的特征向量相等。但只需要计算一个特征向量。速度提高了 5% ~15%。

第六步：同时假如使用之前的特征方程计算，对于可扩展曲面和接近于平面的曲面来说，计算出来的特征向量的特征值也是零，因此需要进行特殊处理，而转换后的特征方程没有这样的问题，永远返回最优解。

12.3　算法推导

频谱参数化的理论基础和 DCP、LSCM 一样。因此得到的是一样的线性系统，但采用了新的方法求解这个线性系统。DCP、LSCM 用固定顶点，从而约束线性系统得到唯一解。在参数化所用到的线性系统是从能量函数求导得到的，因此求解线性系统最终是期望得到能量函数的最小值。频谱方法回归到能量函数本身，采用线性系统矩阵的菲德尔向量来得到能量函数的最小值，从而避免了两个固定顶点约束条件带来的误差。

从前面章推导的 DCP、LSCM 线性方程过程可以得知。

第一步：根据 12.2 节狄利克雷能量的定义为：

$$E_D = \frac{1}{2} \int_\chi |\nabla u|^2 \mathrm{d}A \tag{12-6}$$

第二步：参数化后的三维模型面积计算为：

$$A(u) = \int_\chi \det(u) \mathrm{d}A \tag{12-7}$$

第三步：狄利克雷能量的下限值值就是此面积：

$$E_D \geqslant A(u) \tag{12-8}$$

第四步：保角度参数化映射定义的保角能量为：

$$E_C(u) = E_D(u) - A(u) \tag{12-9}$$

第五步：此能量为零，那么就得到一个保角映射。

第六步：狄利克雷能量离散化后变为：

$$E_D(u) = \sum_{e_{ij}} \frac{1}{4} (\cot(\theta_{ij}) + \cot(\theta_{ji}))(u_i - u_j)^2 \tag{12-10}$$

第七步：矩阵形式表示为：

$$E_D(u) = \frac{1}{2}u^t L_D u \tag{12-11}$$

第八步：面积计算离散化为：

$$A_T(u) = \sum_{e_{ij} \in T} \frac{1}{2}(u_i v_j - u_j v_i) \tag{12-12}$$

第九步：用∂u表示三维模型的边界，所有面积之和可以只是用边界上的顶点进行计算：

$$A(u) = \sum_{e_{ij} \in \partial u} \frac{1}{2}(u_i v_j - u_j u_i) \tag{12-13}$$

第十步：用矩阵形式表示为：

$$A(u) = \frac{1}{2}u^t A u \tag{12-14}$$

第十一步：最后离散化的保角能量$E_C = E_D - A$，矩阵形式为：

$$E_C(u) = \frac{1}{2}u^t L_C u \tag{12-15}$$

第十二步：其中$L_C = L_D - A$是一个稀疏、对称矩阵。能量函数矩阵$E_C(u)$求导得到一个线性方程组，此方程组的解使能量函数达到最小值，也就是保角参数化的结果。

$$L_C u = 0$$

第十三步：通过固定两个顶点可以避免这个线性方程组的退化解，也就是等于常数的解。也确定了未知数的平移、旋转、缩放。根据12.2节的实验显示，选择不同的两个点，得到的参数化平面不一样。因此固定两个点会带来扭曲。

第十四步：假如不选择固定的顶点来约束优化问题，那么必须加上其他的约束条件。固定顶点的约束条件是加在单个顶点上的，因此会造成这两个固定顶点周围的扭曲很大，从而引起全局的扭曲。

第十五步：在上述能量函数优化问题的数学系统的构造中，具有位移、缩放、旋转自由度。因此，可以选择把约束条件设置为所有展开的参数化顶点位置中心为零，这样就移除了平移自由度。其次把所有顶点到中心的距离平方和设置为1。这样就移除了缩放自由度。旋转自由度不需要移除，因为参数化展开后任意方向和角度旋转都是等价的解。而固定顶点的方法由于也固定了旋转的自由度，从而造成扭曲比较大。

第十六步：因此整个参数化的数学模型变为具有如下的约束条件的优化问题：

$$u^* = \arg\min_u u^t L u$$

$$u'e = 0$$

$$u'u = 1$$

第十七步：通过菲德勒向量和优化问题的等价关系可以得出，上述的二次形式优化问题的解，可以等价为矩阵的菲德尔向量，也就是如下的形式。

$$Lu^* = \lambda u^*$$

第十八步：从12.2节可知，由于菲德尔向量是瑞利商的最小值。因此需要使瑞利商的分子最小化，也就是尽可能地保角映射。同时也要使分母最大化，也就是到原点的距离平方和最大。最后达到两者平衡的结果。平衡的结果有时会严重损害保角映射，使局部角度失真，在内部点上也会产生翻转。

$$u^* = \underset{u}{\operatorname{argmin}} \left| \frac{u^t L_C u}{u^t u} \right|$$

第十九步：为了解决这些问题，需要对上述的频谱参数化进行修改。通过采用如下形式的扩展特征向量来得到更好的结果。

$$L_C u = \lambda B u \tag{12-16}$$

第二十步：其中 B 是 $2V \times 2V$ 的对角阵，对角线上只有对应边界上点的项是 1，其他都是零，假如有内部其他边界，相对应的值也是零。因为 L_C 理论上是半正定，L_C 和 B 都是对称的，因此扩展的特征值和特征向量都是实数（Real）。

第二十一步：这个扩展特征向量，对应于如下函数的最小值：

$$\begin{aligned} u^* &= \underset{u}{\operatorname{argmin}}\, u^t L_C u \\ u^t B e &= 0 \\ u^t B u &= 1 \end{aligned} \tag{12-17}$$

第二十二步：其中 e 是 $2V \times 2$ 矩阵，这个矩阵中的项 $e_{i,1}$ 的值是 1，其余项的值是零。从而扩展特征向量对应的改进的瑞利商是：

$$\frac{u^t L_C u}{u^t B u} \tag{12-18}$$

第二十三步：式（12-18）中，分母只依赖于边界上的点，瑞利商的最小值在保角和边界点距离之和进行平衡，和内部点没有关系，从而得到更好的结果。

第二十四步：很多三维网格模型在平滑的部分采样很少，而在弯曲、细节多的地方采样更多。因此，可以通过把三角形面积的倒数作为权重加到狄利克雷能量函数里面来进行扩展，也就是矩阵 L_D 需要对每项除以相对应的三角形面积。

第二十五步：而矩阵 A 因为内部边不能互相取消，所以在矩阵 A 中，对于每个边 (i,j) 上项目相对应项的值的公式为：

$$\begin{aligned} A_{u_i,v_j} &= \frac{1}{2}\left[\frac{1}{|T_{ijk}|} - \frac{1}{|T_{ijl}|}\right] \\ A_{v_i,u_j} &= \frac{1}{2}\left[-\frac{1}{|T_{ijk}|} + \frac{1}{|T_{ijl}|}\right] \end{aligned} \tag{12-19}$$

第二十六步：这样把初始模型的采样放到参数化构造里面就可以解决由于采样不均匀造成的结果扭曲。这样处理三维模型采样不均匀的方法也可以应用到 LSCM/DCP 参数化算法里面。

12.4　核心代码

频谱参数化通过构建能量和面积矩阵，然后对这个矩阵求解特征向量，从而得到菲德尔向量。再根据菲德尔向量得到参数化的结果。频谱参数化的核心代码包括构造能量函数对应的各种矩阵，以及调用函数库求解特征向量。下面是频谱参数化的核心代码。

（1）构建狄利克雷能量函数生成的矩阵。

```
public SparseMatrix BuildMatrixRigid(TriMesh mesh)
```

```
{
    int n = mesh.Vertices.Count;
    SparseMatrix L = new SparseMatrix(n,n);
    for (int i =0;i < mesh.Faces.Count;i ++)
    {
        int c1 = mesh.Faces[i].GetVertex(0).Index;
        int c2 = mesh.Faces[i].GetVertex(1).Index;
        int c3 = mesh.Faces[i].GetVertex(2).Index;
        Vector3D v1 = mesh.Faces[i].GetVertex(0).Traits.Position;
        Vector3D v2 = mesh.Faces[i].GetVertex(1).Traits.Position;
        Vector3D v3 = mesh.Faces[i].GetVertex(2).Traits.Position;
        double cot1 = (v2 - v1).Dot(v3 - v1)
                    /(v2 - v1).Cross(v3 - v1).Length();
        double cot2 = (v3 - v2).Dot(v1 - v2)
                    /(v3 - v2).Cross(v1 - v2).Length();
        double cot3 = (v1 - v3).Dot(v2 - v3)
                    /(v1 - v3).Cross(v2 - v3).Length();
        L.AddValueTo(c1,c2, - cot3/2);
        L.AddValueTo(c2,c1, - cot3/2);
        L.AddValueTo(c2,c3, - cot1/2);
        L.AddValueTo(c3,c2, - cot1/2);
        L.AddValueTo(c3,c1, - cot2/2);
        L.AddValueTo(c1,c3, - cot2/2);

    for (int i =0;i < n;i ++)
    {
        double sum =0;
        foreach (SparseMatrix.Element e in L.Rows[i])
        {
            sum += e.value;
        }
        L.AddValueTo(i,i, - sum);
    }
    L.SortElement();
    return L;
}
```

（2）由于频谱参数化是把 u、v 坐标混合到一起计算的，因此需要扩展这个矩阵。

```
private SparseMatrix BuildMatrixLd()
{
    SparseMatrix L = LaplaceManager.Instance.BuildMatrixRigid(Mesh);
    int n = Mesh.Vertices.Count;
```

```
    SparseMatrix  Ld = new SparseMatrix(2 * n,2 * n);
    foreach (List < SparseMatrix. Element > row in L. Rows)
    {
        foreach (SparseMatrix. Element rowItem in row)
        {
            int i = rowItem. i;
            int j = rowItem. j;
            double value = rowItem. value;
            Ld. AddValueTo(2 * i,2 * j,value);
            Ld. AddValueTo(2 * i + 1,2 * j + 1,value);
        }
    }
    return Ld;
}
```

(3) 根据三维模型的边界顶点，构建面积矩阵。

```
public  SparseMatrix BuildMatrixArea(TriMesh mesh)
{
    List < List < TriMesh. HalfEdge >> bounds =
        TriMeshUtil. RetrieveBoundaryEdgeAll(mesh);
    int n = mesh. Vertices. Count;
    SparseMatrix A = new SparseMatrix(2 * n,2 * n);
    foreach (List < TriMesh. HalfEdge > oneBoundary in bounds)
    {
        foreach (TriMesh. HalfEdge currentHF in oneBoundary)
        {
            int Vi = currentHF. FromVertex. Index;
            int Vj = currentHF. ToVertex. Index;
            A[2 * Vi,2 * Vj + 1] += -0.5;
            A[2 * Vj + 1,2 * Vi] += -0.5;
            A[2 * Vj,2 * Vi + 1] += 0.5;
            A[2 * Vi + 1,2 * Vj] += 0.5;
        }
    }
    return A;
}
```

(4) 为面积矩阵补充权重。

```
private SparseMatrix BuildMatrixA()
{
    SparseMatrix A = LaplaceManager. Instance. BuildMatrixArea(Mesh);
    foreach (TriMesh. Edge e in Mesh. Edges)
```

```
        {
            if (e.OnBoundary)
            {
                continue;
            }
            TriMesh.HalfEdge Hij = e.HalfEdge0;

            int Vi = Hij.FromVertex.Index;
            int Vj = Hij.ToVertex.Index;

            double areaTijk =
                Math.Abs(TriMeshUtil.ComputeAreaFace(Hij.Face));
            double areaTijl =
                Math.Abs(TriMeshUtil.ComputeAreaFace(Hij.Opposite.Face));

            A[2 * Vi,2 * Vj + 1] = (1/2)
                                * (1/areaTijk - 1/areaTijl);
            A[2 * Vj,2 * Vi + 1] = (1/2)
                                * ( - 1/areaTijk + 1/areaTijl);

        }
        return A;
    }
```

（5）用求解特征向量的方法得到参数化的结果。

```
    public override void Parameterize()
    {
        SparseMatrix Ld = BuildMatrixLd();
        SparseMatrix A = BuildMatrixA();
        SparseMatrix Lc = Ld.Minus(A);
        double[] u = TriMeshFunction.ComputeEigenVector(Lc,0);
        Convert(u);
    }
```

12.5 实验分析

频谱参数化由于不需要固定两个顶点，因此效果比 LSCM 和 DCP 要好。也是三维模型参数化算法在尽量保持角度不变这个限制条件下，能够达到的最好结果。最重要的缺点是需要求解特征向量，这是一个非常耗时的操作。而 LSCM 和 DCP 只需要求解线性方程组。频谱参数化算法最重要的特点不仅是不需要任何固定点，另一个特点是能够处理采样不均匀的三维模型，从而得到更好的展平效果。

（1）如图 12–2 所示是各种形状的模型在频谱参数化算法下的展开和贴图的效果。从中

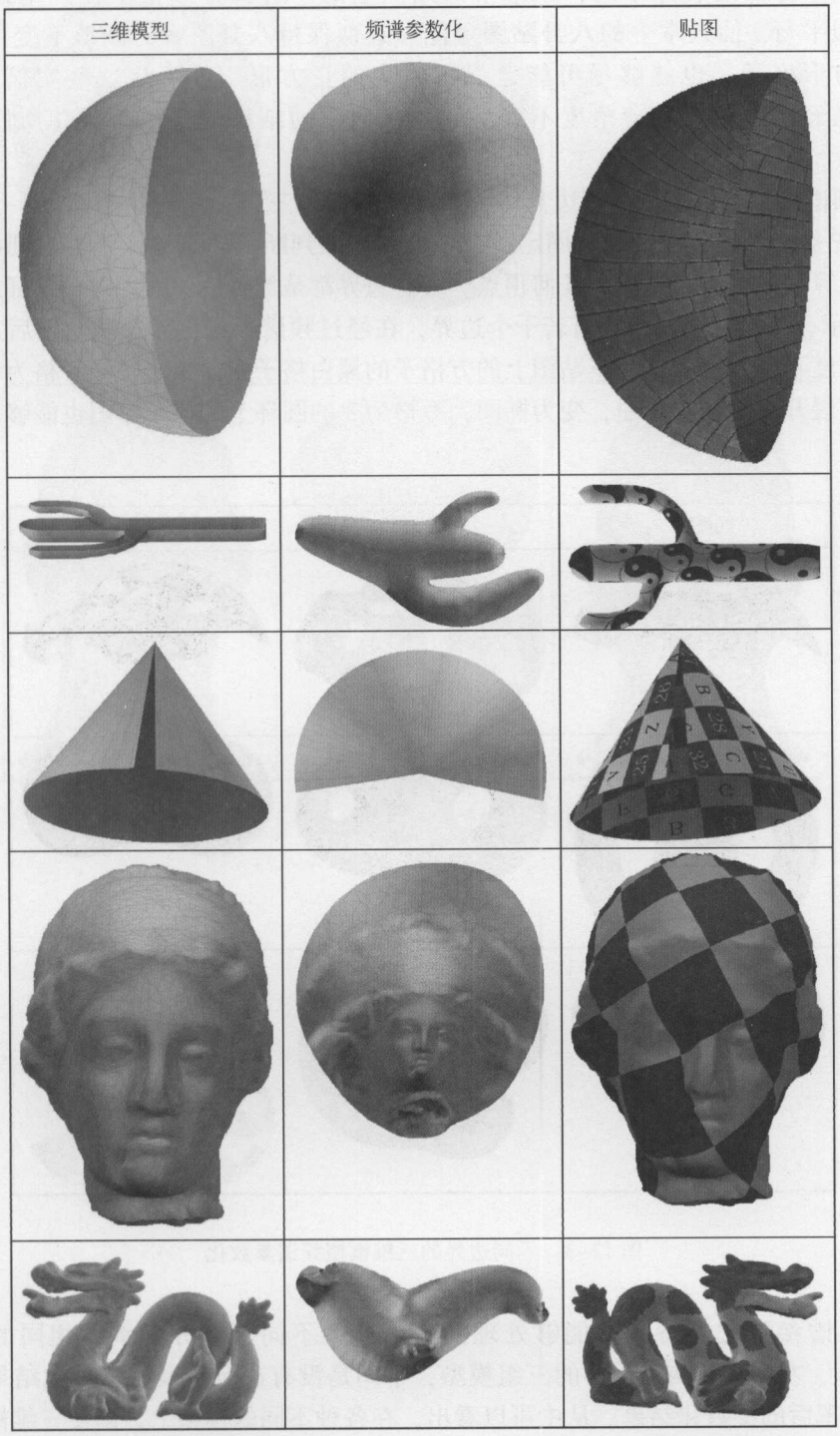


图 12–2　各种不同形状三维模型的频谱参数化

可以看出，假如三维模型和平面不是相差特别远，那么频谱参数化展开后保持角度的效果很好。例如，半球面的展开贴图，原来平面贴图上的平行长方形格子在三维贴图上仍能够保持近似平行。仙人掌上的八卦贴图也能够近似保持八卦图案的圆形不变。圆锥及人脸上的正方形格子，也能够尽可能地保持原来的正方形不变。从这些实例可以看出，频谱算法已经尽可能地保持角度不变。频谱参数化的结果是保角参数化所能够达到的极限。

（2）频谱方法也可以处理多边界的网格模型。假如三维模型的边界多于一个，那么用LSCM、DCP等参数化算法，需要固定两个顶点。很难判断哪两个顶点能够得到最优解。在频谱参数化算法中，不需要固定任何顶点，所有边界都是等价的，因此得到最好的效果。如图12-3所示，每个三维模型都有若干个边界，在经过频谱参数化展开和贴图后，能够尽可能地保持角度不变。例如，汽车贴图上的方格子的黑白格子扭曲很小，而亏格为2圆环上的贴图中的圆展开后稍微有扭曲，变为椭圆，亏格为3的圆环上的格子贴图也能够尽可能保持直角。

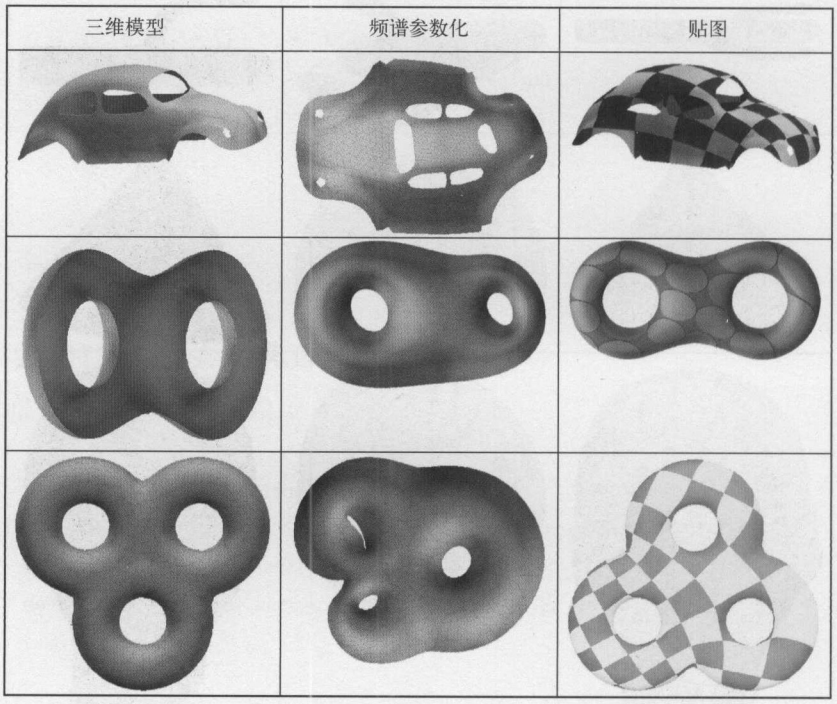


图12-3　不同边界的三维模型频谱参数化

（3）频谱参数化的特点是能够处理同一个模型不同部位分辨率不相同的情况。如图12-4所示，左侧是不均匀采样的三组模型，中间是没有进行扩展的参数化结果，右侧是考虑到采样率后的参数化结果。从中可以看出，在各种不同的模型上，即使三维模型的采样不均匀，频谱参数化算法也都可以很好地处理。

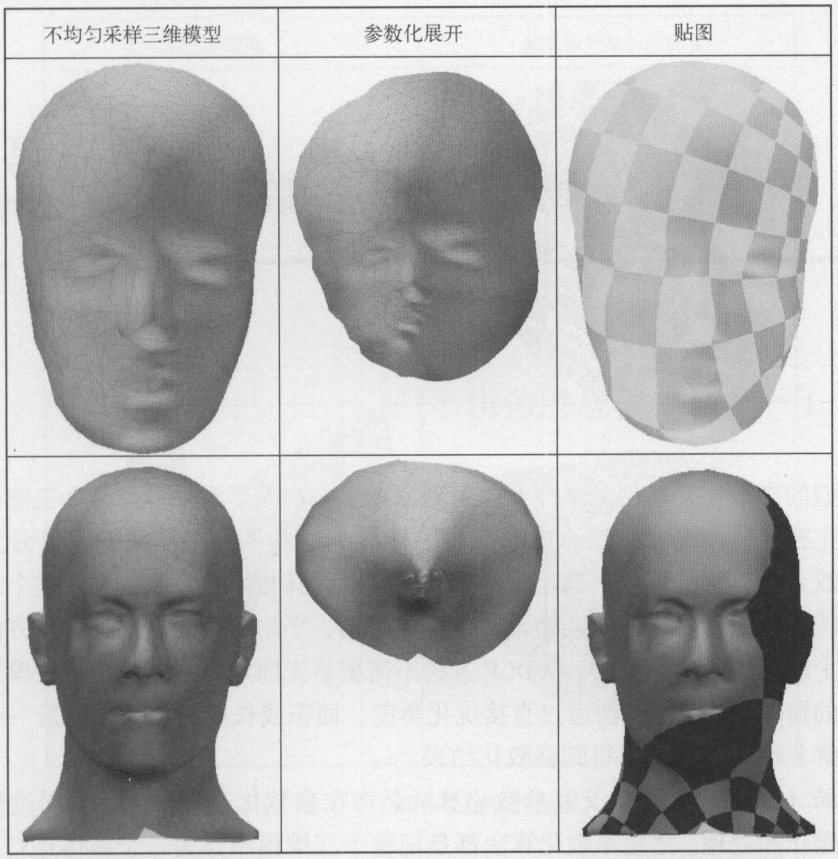


图 12-4　不同分辨率的三维模型频谱参数化

第13章 局部全局参数化算法

13.1 局部全局思想逻辑

三维模型的参数化是指把三维模型映射到二维平面，除了圆柱等可扩展三维曲面之外的模型都会产生扭曲。因此参数化的目标是尽可能减少这些扭曲。扭曲的度量分为面积、角度、边长，或者这三个的组合。通过把一个顶点表示为其相邻的顶点的和，整个参数化可以构建为一个线性系统。权重的方法通常有中值、余切、平均权重等方法。这些方法需要把边界映射到一个凸多边形上。LSCM 和 DCP 方法不需要事先固定好边界，但有些模型映射后会造成面翻转的情况。另一种方法通过直接优化角度、面积或长度的扭曲来构造一个非线性系统，通过求解非线性系统可以得到参数化结果。

局部全局（Local/Global）交互参数化算法是指在参数化过程中，从全局进行思考，但从局部进行操作和处理。之前参数化算法都是把整个三维模型作为一个整体进行参数化。从整体的角度进行设计算法。但三维模型是由若干三角形构成的，每个三角形都可以没有任何扭曲地映射到平面，这种局部全局的参数化算法思路是先把每个三角形无扭曲地映射到平面，然后再调整三角形的形状，使三角形满足某种条件。每个三角形在调整形状时目标是减少扭曲，但又必须满足构成一个合法的三维模型。根据调整三角形形状时的标准，又分为尽可能相似的参数化 ASAP（As - Similar - As - Possible）和尽可能刚性的参数化 ARAP（As - Rigid - As - Possible）。ASAP 参数化等价于 LSCM，也就是 ASAP 参数化算法设计的推导起点和思路与 LSCM 不一样，但最终的参数化结果是一样的。

虽然三维模型上每个单独的三角形可以没有扭曲、独立地映射到二维平面，但这样映射得到结果是每个三角形和其他的三角形就无法连接到一起。局部全局算法就是先把每个三角形没有扭曲地映射到二维平面，然后再构造一个系统把所有在二维平面上的三角形连接到一起，如图 13-1 所示。

在连接这些二维平面上的三角形时，不是所有的三角形都能够保持形状不变，很多三角形需要进行变换。如果这些三角形只发生相似（Similarity Transform）变换的话，那么结果是保角参数化。如果只发生旋转变换（Rotation Transform）的话，结果是等距变换，如果等距变换矩阵的秩是单位大小（Unit），那么变换是等积变换。因此，通过限制变换矩阵，可以得到相应的参数化结果。

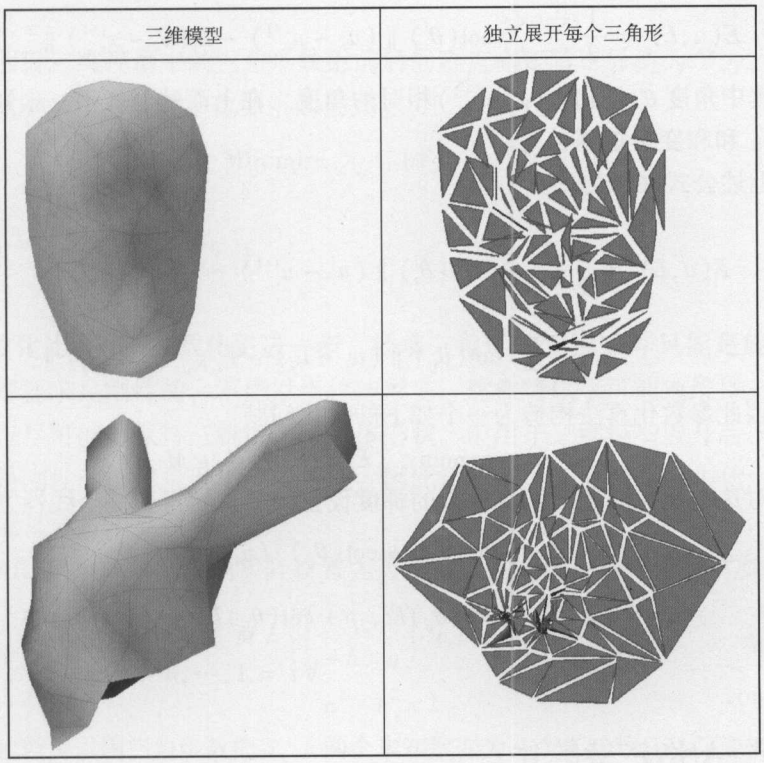


图 13-1　三角形单独展开图示

13.2　算法设计

ASAP 和 ARAP 参数化算法的设计也需要构造一个能量函数，只是现在这个能量函数的构造来源于变换矩阵。也就是需要限制每个三角形的变换矩阵不能是任意的矩阵，对于 ARAP 算法来说，变换矩阵只能发生旋转变换，对于 ASAP 算法来说，变换矩阵只能发生相似变换。能量函数定义在当前每个三角形的变换矩阵上，也就是希望得到每个三角形的变换矩阵尽可能是相似变换，或者旋转变换，那么就可以定义一个基于矩阵的能量函数。能量函数是基于矩阵的，矩阵本身是基于映射前和映射后的顶点位置，映射前的顶点位置是已知的，映射后的顶点位置是未知的，从而能量函数就是映射后顶点位置的函数。有了能量函数之后，再对能量函数求导得到最小值，就得到所对应的映射后的顶点位置，也就是参数化的结果。

第一步：假设三维模型上一个三角形 t 表示为 $x_t = \{x_t^0, x_t^1, x_t^2\}$，相应的二维平面上的三角形表示为 $u_t = \{u_t^0, u_t^1, u_t^2\}$。那么三维和二维的两个三角形 x_t、u_t 的关系可以用一个 2×2 的雅克比（Jobabian）矩阵 $J_t(u)$来表示。

第二步：如果用 L_t 来表示受到限制的变换矩阵，也就是限制为相似变换、旋转变换，那么可以定义如下的能量函数，这个函数表示了可能的变换和目标变换之间的差别。

$$E(u,L) = \sum_{t=1}^{T} A_t \| J_t(u) - L_t \|_F^2 \tag{13-1}$$

第三步：其中 A_t 是三角形面积。上面能量函数可以重构为：

$$E(u,L) = \frac{1}{2}\sum_{t=1}^{T}\sum_{i=0}^{2}\cot(\theta_t^i)\,\|(u_t^i - u_t^{i+1}) - L_t(x_t^i - x_t^{i+1})\|^2 \tag{13-2}$$

第四步：其中角度 θ_t^i 是和边 (x_t^i, x_t^{i+1}) 相对的角度。在上面的公式里，未知数为二维平面上的映射顶点 u 和和变换矩阵 L。

第五步：上述公式展开为：

$$\begin{aligned} E(u,L) &= \frac{1}{2}\sum_{t=1}^{T}\sum_{i=0}^{2}\cot(\theta_t^i)\,\|(u_t^i - u_t^{i+1}) - L_t(x_t^i - x_t^{i+1}\|^2 \\ &= \frac{1}{2}\sum_{(i,j)\in he}\cot(\theta_{ij})\,\|(u_i - u_j) - L_{t\,(i,j)}(x_i - x_j)\|^2 \end{aligned} \tag{13-3}$$

第六步：因此参数化算法构造为一个如下的优化问题：

$$(u,L) = \mathrm{argmin}_{(u,L)}E(u,L), \quad L_t \in M \tag{13-4}$$

第七步：对这个优化问题的构造函数的梯度设置为零，就得到一个线性方程组，如下：

$$\begin{aligned} &\sum_{j\in N(i)}[\cot(\theta_{ij}) + \cot(\theta_{ij})](u_i - u_j) \\ &= \sum_{j=N(i)}[\cot(\theta_{ij})L_{t(i,j)} + \cot(\theta_{ji})L_{t_{(j,i)}}](x_i - x_j) \\ &\qquad \forall i = 1,\cdots,n \end{aligned} \tag{13-5}$$

13.3 ASAP 参数化

13.2 节中构造的能量函数里的未知数是每个三角形的变换矩阵。因此整个能量函数与设定的限制变换矩阵有关。根据变换矩阵的不同得到不同的结果。在 ASAP 参数化中，把变换矩阵限制为相似变换矩阵，也就是每个三角形只能做相似变换。每个三角形假如只是单独展开到平面，那么可以无条件地满足只能是相似变换这个限制条件。但由于每个三角形必须和相邻的三角形连接，这个限制造成了无法保证每个三角形都能够具有相似变换，只能是尽可能地进行相似变换。从而定义的能量函数求导就得到整个三维模型上所有三角形变换的最优的相似变换。

第一步：ASAP 参数化目标是获得保角参数化结果，因此变换矩阵 L 限制为相似变换，也就是矩阵限制为如下形式。

$$M = \left\{\begin{pmatrix} a & b \\ -b & a \end{pmatrix} : a,b \in R\right\} \tag{13-6}$$

第二步：其中的未知数变量是：

$$a = (a_1,\cdots,a_T)$$
$$b = (b_1,\cdots,b_T)$$

第三步：把上述矩阵代入 13.2 节的线性方程组，可以得到一个线性系统，这个系统是以三维顶点的映射坐标 u 和 a、b 为未知数变量的线性系统，其中 a、b 虽然是未知数的变量，但只是辅助性的未知数变量。

第四步：线性系统可以用最小二次方求解。在求解的结果中，包含 u、a、b，但只需要 u 来进行参数化，也就是设置每个顶点相应的映射位置，a 和 b 不需要。

第五步：这样设计的线性系统和 LSCM 一样具有位移和旋转两个自由度，因此也需要固

定两个顶点。

第六步：假设三维模型中的三角形映射前后的雅克比矩阵奇异值为 Γ、γ，和 LSCM 的结果一样，那么 ASAP 相当于最小化如下的公式。

$$\text{Minimize} \sum_t A_t (\Gamma_t - \gamma_t)^2$$

13.4　ARAP 参数化

ARAP 参数化比 ASAP 参数化更进一步，要求三角形的变换矩阵只能是旋转矩阵，也就是所有的三角形要求是刚性的，不能发生拉伸形变。这样可以尽可能地维持三角形原来的形状，从而就可以尽可能地保持三维模型的整体形状。但由于三维模型展开后不可能完全满足所有三角形都是刚性的，都只发生旋转变换，因此需要把误差或扭曲最小化。也就是在变化矩阵上定义的能量函数求导得到极值。

第一步：ARAP 参数化把限制矩阵设置为等距变换矩阵，变换矩阵形式如下。

$$M = \left\{ \begin{pmatrix} a & b \\ -b & a \end{pmatrix} : a, b \in R \right\}$$

$$a^2 + b^2 = 1$$

第二步：也就可以用三角函数把 a、b 两个未知数变为一个未知数，从而变换矩阵形式如下。

$$M = \left\{ \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} : \theta \in [0, 2\pi) \right\} \tag{13-7}$$

第三步：虽然和 ASAP 相比，ARAP 现在只有一个未知数，但 cos、sin 使这个未知数变为非线性的，因此无法用线性系统求解。

第四步：但可以用迭代的方法求解，也就是先求出变换矩阵 L，然后再求解 U，再进行迭代，直到系统收敛。

第五步：已知一个初始化的 U，通常可以用 ASAP 或 LSCM 的结果作为初始值，求解变换矩阵 L。

第六步：对于如下 2×2 的矩阵：

$$S_t(u) = \sum_{i=0}^{2} \cot(\theta_t^i)(u_t^i - u_t^{i+1})(x_t^{i+1} - x_t^{i+1})^{\mathrm{T}} \tag{13-8}$$

第七步：可以做 SVD 分解为：

$$J = U \sum V^{\mathrm{T}} \tag{13-9}$$

第八步：其中

$$\sum = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$$

第九步：那么可以得到最优的变换矩阵为：

$$L = UV^{\mathrm{T}} \tag{13-10}$$

第十步：通过上一步的 L，把 L 代入下面的线性系统中，求解 U：

$$\sum_{j\in N(i)}[\cot(\theta_{ij})+\cot(\theta_{ji})](u_i-u_j)$$
$$=\sum_{j\in N(i)}[\cot(\theta_{ij})L_{t(i,j)}+\cot(\theta_{ji})L_{t(j,i)}](x_i-x_j) \tag{13-11}$$
$$\forall i=1,\cdots,n$$

第十一步：迭代上述两步，直到 U 没有变化为止。

第十二步：在这个线性系统中，左边的矩阵保持不变，每次迭代变得是右边的向量，因此可以对左边的矩阵进行预分解，从而不需要每次迭代都进行分解。

第十三步：假设三维模型中的三角形映射前后的雅克比矩阵奇异值为 Γ、γ，那么 ARAP 相当于最小化如下的公式：

$$\text{Minimize}\sum_t A_t\lfloor(\Gamma-1)^2+(\gamma-1)^2\rfloor$$

13.5 混合算法

ASAP 的算法目标是保持角度尽可能不变，属于保角算法。但 ASAP 算法保持面积的效果不好。在各种保角算法中，ABF 算法是在保持角度的前提下，最能够保持面积尽可能不变。ARAP 算法保面积的效果很好，但保角的效果受到影响。因此，可以把 ASAP 和 ARAP 结合起来，这样的算法称为混合算法（Hybrid）。

混合算法定义一个新的能量函数如下：

$$E(u,a,b)=\frac{1}{2}\sum_{t=1}^{T}\left[\sum_{i=0}^{2}\cot(\theta_t^i)\parallel\nabla e_t^i\parallel^2+\lambda(a_t^2+b_t^2-1)^2\right] \tag{13-12}$$

其中：

$$\nabla e_t^i=(u_t^i-u_t^{i+1})-\begin{pmatrix}a_t & b_t\\ -b_t & a_t\end{pmatrix}(x_t^i-x_t^{i+1})$$
$$\lambda\in[0,\infty) \tag{13-13}$$

参数 $\lambda=0$ 时就是 ASAP，如果 λ 的值很大，就相当于 ARAP。

效果度量。可以用三维模型上的三角形和二维平面上参数化后的三角形的变换矩阵的特征值来对参数化进行度量。公式如下：

$$S_t(u)=\sum_{i=0}^{2}\cot(\theta_t^i)(u_t^i-u_t^{i+1})(x_t^i-x_t^{i+1})^{\mathrm{T}} \tag{13-14}$$

上述 2×2 矩阵进行 SVD 分解后，

$$D^{\text{angle}}=\sum_t\rho_t(\sigma_t^1/\sigma_t^1+\sigma_t^2/\sigma_t^1)$$
$$D^{\text{area}}=\sum_t\rho_t(\sigma_t^1\sigma_t^2+1/(\sigma_t^1\sigma_t^2)) \tag{13-15}$$

其中

$$\rho_t=A_t/\sum_t A_t \tag{13-16}$$

13.6 ASAP 算法代码

ASAP 算法首先需要把三维模型上的每个三角形独立映射到平面，然后再计算每个三角形的变换矩阵，最终得到三维模型的顶点坐标。由于变换矩阵受到约束条件限制，因此可以

和顶点坐标同步进行计算。ASAP 核心代码包含映射三角形函数，构造线性系统矩阵函数，以及构造线性系统右边向量等函数。

1. 把三维模型上的所有三角形独立映射到平面上

这个函数把三维模型在三维空间上的每个三角形都独立映射到同一个平面上。映射完成后，每个在平面上的三角形和原来三维空间上的三角形大小和形状都一样，只是位置发生了变化。原来三维空间里的三角形是互相连接的，映射到平面后的三角形是互不关联的。

```
public Vector2D[ ] MapTriangleToPlane(TriMesh Mesh)
{
    Vector2D[ ] vectors = new Vector2D[ Mesh. HalfEdges. Count ];
    foreach(TriMesh. Face face in Mesh. Faces)
    {
        TriMesh. HalfEdge hf1 = face. GetHalfedge(0);
        TriMesh. HalfEdge hf2 = face. GetHalfedge(1);
        TriMesh. HalfEdge hf3 = face. GetHalfedge(2);
        double hf1Length = (hf1. ToVertex. Traits. Position
            - hf1. FromVertex. Traits. Position). Length( );
        double hf2Length = (hf2. ToVertex. Traits. Position
            - hf2. FromVertex. Traits. Position). Length( );
        double hf3Length = (hf3. ToVertex. Traits. Position
            - hf3. FromVertex. Traits. Position). Length( );

        double hf13Angle = Math. Acos((hf1Length * hf1Length
                                    + hf3Length * hf3Length
                                    - hf2Length * hf2Length)
                                /(2 * hf1Length * hf3Length));

        Vector2D UPosition1 = new Vector2D(0,0);
        Vector2D UPosition2 = new Vector2D(hf1Length,0);
        Vector2D UPosition3 = new Vector2D(hf3Length * Math. Cos(hf13Angle),
                                    hf3Length * Math. Sin(hf13Angle));

        Vector2D hf1Mapping = UPosition2 - UPosition1;
        Vector2D hf2Mapping = UPosition3 - UPosition2;
        Vector2D hf3Mapping = UPosition1 - UPosition3;

        vectors[ hf1. Index ] = hf1Mapping;
        vectors[ hf2. Index ] = hf2Mapping;
        vectors[ hf3. Index ] = hf3Mapping;
    }
    return vectors;
}
```

2. 得到固定的两个顶点

任选两个边界上的顶点作为线性系统的约束条件。

```
boundary = new ParaBoundary(Mesh);
List<TriMesh.Vertex> fixedVertice = boundary.GetFixVertice();
```

3. 对边界进行映射

```
int[] HFold2New = new int[Mesh.HalfEdges.Count];
int sum = 0;
foreach(TriMesh.HalfEdge hf in mesh.HalfEdges)
{
    if(!hf.OnBoundary)
    {
        HFold2New[hf.Index] = sum;
        sum ++;
    }
}
```

4. 构造矩阵

构造大小为 $2m \times (2v + 2f)$ 的矩阵，其中 m 是内部半边数量，v 是顶点数量，f 是面的数量。

```
int row = 2 * sum;
int col = 2 * Mesh.Vertices.Count + 2 * Mesh.Faces.Count;
int faceOffset = 2 * Mesh.Vertices.Count;
SparseMatrix L = new SparseMatrix(row,col);
```

5. 构建线性系统的矩阵

根据能量函数求导得到的线性方程组来构造方程组左边的矩阵。

```
foreach(TriMesh.HalfEdge hij in mesh.HalfEdges)
{
    if(!hij.OnBoundary)
    {
        int hfindex = HFold2New[hij.Index];

        TriMesh.Vertex Vj = hij.ToVertex;
        TriMesh.Vertex Vi = hij.FromVertex;
        double cotIJ = 0;
        double hfLength = TriMeshUtil.ComputeEdgeLength(hij.Edge);
        double cLength = TriMeshUtil.ComputeEdgeLength(hij.Next.Edge);
        double aLength = TriMeshUtil.ComputeEdgeLength(hij.Previous.Edge);
        double angleIJ = Math.Acos((aLength * aLength
                                    + cLength * cLength
                                    - hfLength * hfLength)
                                   /(2 * aLength * cLength));
```

```
            cotIJ = 1/Math. Tan( angleIJ) ;

            L[2 * hfindex,2 * Vi. Index] + = cotIJ;
            L[2 * hfindex,2 * Vj. Index] - = cotIJ;

            L[2 * hfindex + 1,2 * Vi. Index + 1] + = cotIJ;
            L[2 * hfindex + 1,2 * Vj. Index + 1] - = cotIJ;

            Vector2D edgeIJ = edgeVectors[ hij. Index] ;
            TriMesh. Face faceIJ = hij. Face;
            int r = 2 * hfindex;
            int c = faceOffset + 2 * faceIJ. Index;
            L[r,c] - = cotIJ * edgeIJ. x;
            L[r,c + 1] - = cotIJ * edgeIJ. y;
            L[r + 1,c] - = cotIJ * edgeIJ. y;
            L[r + 1,c + 1] + = cotIJ * edgeIJ. x;

        }
    }
```

6. 增加固定顶点

```
    foreach( TriMesh. Vertex vertex in fixedVertice)
    {
            L. AddRow( ) ;
            L[ L. Rows. Count - 1,2 * vertex. Index] = 1;

            L. AddRow( ) ;
            L[ L. Rows. Count - 1,2 * vertex. Index + 1] = 1;
    }
```

7. 构建线性系统右边向量

```
    private double[ ] BuildPartB( int size, List < TriMesh. Vertex > fixedVertice)
    {
        double[ ] bPart = new double[ size] ;
        for( int i = 0;i < bPart. Length;i ++ )
        {
        bPart[ i] = 0;
        }
        int start = bPart. Length - fixedVertice. Count * 2;
        bPart[ start] = boundary. handleFirst. x;
        bPart[ start + 1] = boundary. handleFirst. y;
```

```
        bPart[start + 2] = boundary.handleSecond.x;
        bPart[start + 3] = boundary.handleSecond.y;
        return bPart;
    }
```

8. 用最小二乘法求解

```
    public override void Parameterize()
    {
        Vector2D[] vectors = MapTriangleToPlane(Mesh);
        boundary = new ParaBoundary(Mesh);
        List<TriMesh.Vertex> fixedVertice = boundary.GetFixVertice();
        SparseMatrix sparseMatrix = BuildLeftMatrix(Mesh, vectors, fixedVertice);
        double[] partB = BuildPartB(sparseMatrix.Rows.Count, fixedVertice);
        double[] unKnownUV =
            LinearSystem.Instance.SloveLeastSquare(ref sparseMatrix, ref partB);
        Convert(unKnownUV);
    }
```

13.7 ARAP 算法代码

ARAP 算法需要用迭代的方法进行求解，从而可以得到更好的效果。首先需要一个初始化的参数化坐标。这个初始化参数化坐标可以用任意其他的算法得到，如 LSCM、DCP、ASAP 等。然后根据这个初始化的坐标进行迭代，直到每次能量函数的改变不超过定义的阈值才终止迭代。每次迭代线性系统左边矩阵不变，改变的是线性系统右边的向量。

（1）用 LSCM 算法得到初始的参数化坐标作为初始值。

```
    private static Vector2D[] InitUV(TriMesh mesh)
    {
        ParameterizationLSCM lscm = new ParameterizationLSCM(mesh);
        lscm.Parameterize();
        Vector2D[] vt = new Vector2D[mesh.Vertices.Count];
        foreach(var v in mesh.Vertices)
        {

            vt[v.Index] = new Vector2D(lscm.UnknownU[v.Index],
                                       lscm.UnknownV[v.Index]);
        }
        return vt;
    }
```

（2）把三维模型每个面展开到平面上，这一步计算的展开保持原来三角形每个面的大小完全不变，但各个面在平面上会互相覆盖。

```
public Vector2D[ , ] CalEdgeVectors(TriMesh mesh)
{
    Vector2D[ , ] vectors = new Vector2D[ mesh. Faces. Count,3 ];
    foreach(TriMesh. Face face in mesh. Faces)
    {
        Vector3D[ ] arr = GetVertices(face,p => p. Traits. Position);
        double a = (arr[0] - arr[1]). Length();
        double b = (arr[1] - arr[2]). Length();
        double c = (arr[2] - arr[0]). Length();
        double angle = Math. Acos((a * a + c * c - b * b)/(2 * a * c));
        Vector2D UPosition1 = new Vector2D(0,0);
        Vector2D UPosition2 = new Vector2D(a,0);
        Vector2D UPosition3 = new Vector2D(c * Math. Cos(angle),
                                           c * Math. Sin(angle));

        Vector2D EdgeVectorsItem1 = UPosition3 - UPosition2;
        Vector2D EdgeVectorsItem2 = UPosition1 - UPosition3;
        Vector2D EdgeVectorsItem3 = UPosition2 - UPosition1;

        vectors[face. Index,0] = EdgeVectorsItem1;
        vectors[face. Index,1] = EdgeVectorsItem2;
        vectors[face. Index,2] = EdgeVectorsItem3;
    }
    return vectors;
}
```

(3) 根据当前参数化的结果计算能量函数。

```
public double CalRigidEnergy(TriMesh mesh,Vector2D[ , ] EV,
                Vector2D[ ] vt,double[ , ] C,Matrix2D[ ] R)
{
    double E = 0;
    foreach(TriMesh. Face face in mesh. Faces)
    {
        int[ ] arr = GetVertices(face,p => p. Index);
        for(int i = 0;i < 3;i ++)
        {
            int n = (i + 1)%3;
            int p = (i + 2)%3;
            Vector2D Uij = vt[arr[p]] - vt[arr[n]];
            Vector2D Eij = EV[face. Index,i];
            E + = C[face. Index,i] * (Uij - R[face. Index] * Eij). LengthSquared;
        }
```

```
        }
        return E;
    }
```

（4）对于三维模型的每个三角形，计算参数化之前和参数化之后的变换矩阵。

```
    private Matrix3D ComputeTransformMatrix(Vector2D[] vt,Vector2D[,] EV,
                                            double[,] C,HalfEdgeMesh.Face face)
    {
        Vector2D[] u = GetVertices(face,p => vt[p.Index]);
        Matrix3D uu = new Matrix3D();
        Matrix3D xx = new Matrix3D();
        Matrix3D cc = new Matrix3D();
        for(int i =0;i <3;i ++)
        {
            int n = (i +1)%3;
            int p = (i +2)%3;
            uu[i,0] = u[p].x - u[n].x;
            uu[i,1] = u[p].y - u[n].y;
            xx[i,0] = EV[face.Index,i].x;
            xx[i,1] = EV[face.Index,i].y;
            cc[i,i] = C[face.Index,i];
        }
        Matrix3D CovMat = xx.Transpose() * cc * uu;
        return CovMat;
    }
```

（5）对于一个三角形的变换矩阵用 SVD 分解方法计算这个变换矩阵中的旋转部分。

```
    private static Matrix3D SVDFactorize(Matrix3D CovMat)
    {
        SVD2 svd = new SVD2(CovMat);
        Matrix3D rot = svd.V * svd.U.Transpose();
        if(rot.Det() <0)
        {
            if(svd.E[0] < svd.E[1])
            {
                for(int i =0;i <2;i ++)
                {
                    svd.U[i,0] = -1 * svd.U[i,0];
                }
            }
            else
            {
```

```
                for(int i =0;i <2;i ++)
                {
                    svd. U[i,1] = -1 * svd. U[i,1];
                }
            }
            rot = svd. V * svd. U. Transpose();
        }
        return rot;
    }
```

(6) 计算所有三角形的旋转矩阵。

```
    public Matrix2D[ ] ARAPLocal(TriMesh mesh,Vector2D[ ] vt,
                        Vector2D[ ,] EV,double[ ,] C)
    {
        Matrix2D[ ] R = new Matrix2D[mesh. Faces. Count];
        foreach(var face in mesh. Faces)
        {
            Matrix3D CovMat = ComputeTransformMatrix(vt,EV,C,face);

            Matrix3D rot = SVDFactorize(CovMat);
            R[face. Index] = new Matrix2D(rot[0,0],rot[0,1],
                                          rot[1,0],rot[1,1]);
            }
            return R;
        }
```

(7) 根据新的旋转矩阵，更新线性方程组右边向量，然后再去接线性方程组，得到新的参数化结果。

```
    public Vector2D [ ] ARAPGlobal(TriMesh mesh,Vector2D[ ,] EV,
                    SparseMatrix L,double[ ,] C,Matrix2D[ ] R)
    {
        int l = mesh. Vertices. Count;
        double[ ] bx = new double[l];
        double[ ] by = new double[l];
        foreach(var face in mesh. Faces)
        {
            int[ ] arr = GetVertices(face,p =>p. Index);
            for(int i =0;i <3;i ++)
            {
                int n = (i +1)%3;
                int p = (i +2)%3;
                Vector2D Eij = EV[face. Index,i];
```

```
                Vector2D b = R[face.Index] * Eij * C[face.Index,i];
                bx[arr[n]] -= b.x;
                bx[arr[p]] += b.x;
                by[arr[n]] -= b.y;
                by[arr[p]] += b.y;
            }
        }

        double[] Uy = LinearSystem.Instance.SloveLeastSquare(ref L,ref by);
        double[] Ux = LinearSystem.Instance.SloveLeastSquare(ref L,ref bx);

        Vector2D[] vt = new Vector2D[mesh.Vertices.Count];
        for(int i=0;i<1;i++)
        {
            vt[i].x = Ux[i];
            vt[i].y = Uy[i];
        }
        return vt;
    }
```

（8）用迭代的方法求得最优解。

```
    public Vector2D[] ParameterizeUV(TriMesh mesh)
    {
        Vector2D[] vt = InitUV(mesh);
        Vector2D[,] EV = CalEdgeVectors(mesh);
        double[,] C = CalCots(mesh);
        SparseMatrix L = Laplacian(mesh,C);

        double E = -1;
        double Epre = 0;
        int iterations = 0;
        while(Math.Abs(Epre - E) > 0.0001)
        {
            iterations++;
            Epre = E;
            Matrix2D[] R = ARAPLocal(mesh,vt,EV,C);
            vt = ARAPGlobal(mesh,EV,L,C,R);
            E = CalRigidEnergy(mesh,EV,vt,C,R);
        }
        return vt;
    }
```

13.8　效果分析

（1）ASAP 的效果和 LSCM 的效果一样，能够把各种不同形状的三维模型展开，并贴图。如图 13-2 所示半球、贝多芬头像、圆柱等有边界的三维模型，都可以成功地进行展开。在半球模型贴图之后，可以看出原来二维图像上的正方形角度仍尽可能地保持不变。在贝多芬头像的贴图上可以看出，原来的长方形被拉伸了，但尽可能保持长方形的角度是 90°，也就是保持长方形边的夹角不变。对于圆柱三维模型的展开和贴图来说，由于圆柱模型本身面数比较少，造成拓扑结构不好，因此展开后贴图效果不是很好。

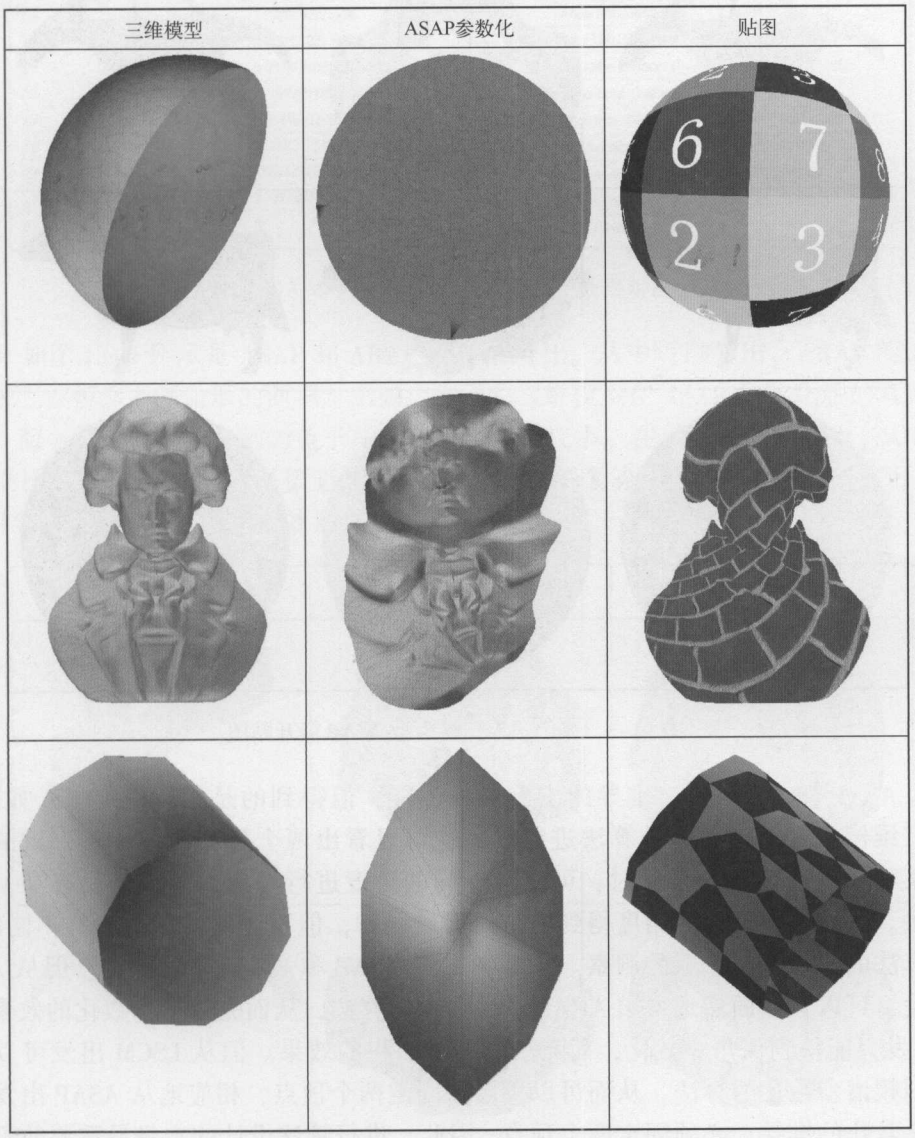


图 13-2　ASAP 参数化展开图示

（2）ASAP 也可以处理具有多个边界的三维模型。如图 13-3 所示，从几种不同的具有多个边界的三维模型展开和贴图效果来看，可以得出对于多边界的模型，ASAP 能够尽可能地保持角度不变。例如，二维图像上的圆形在对有两个边界的半球三维模型贴图之后，仍能够保持圆形，只是每个圆形的大小有变化。而对于形状比较复杂、与平面相差太远的三维模型，如牛的三维模型，部分圆形变成了椭圆。对于具有更多边界的汽车三维模型，ASAP 能够很好地展开，在贴图之后也能够很好地保持角度不变。从牛的 ASAP 展开图可以看出，和 LSCM 一样，ASAP 边界上也有全局相交的部分。

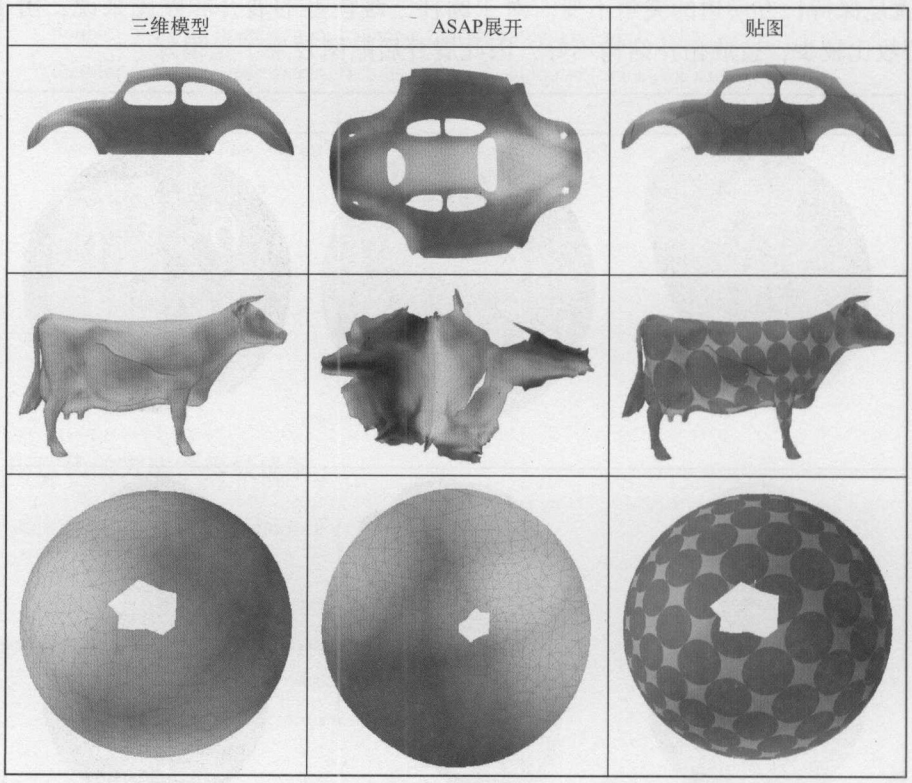
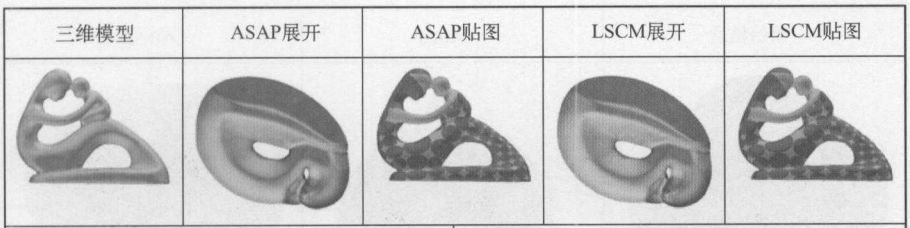


图 13-3　多个边界三维模型 ASAP 展开贴图

（3）ASAP 和 LSCM 虽然推导的出发点不一样，但得到的是相同的结果，如图 13-4 所示，三维模型分别用这两种算法进行展开，可以看出两个算法的结果是一致的。因此在针对某种问题进行算法设计时，可以从不同的角度进行思考，而不是只有唯一一个正确的方法。虽然可能不同的角度得到的结果殊途同归，但不同的角度可以有不同的启发。从而对算法的设计有更深入的洞察。例如，虽然 ASAP 和 LSCM 是一致的，但从 ASAP 的角度出发，可以自然而然地得出 ARAP 的保面积的算法，从而完善了参数化的效果。而从 LSCM 出发只能得到保角参数化，而无法得到保面积的效果。但从 LSCM 出发可以自然而然地得到频谱参数化的算法，从而可以不需要固定两个顶点，相应地从 ASAP 出发无法得到频谱参数化的推导，必须固定两个顶点。因此，进行算法设计的本身最重要的不仅是算法设计的结果，更重要的是算法设计的过程。通过算法设计的过程加深对算法和所涉及的问题本身的洞察和理解。



ASAP扭曲度量

Index	Distortion Name	Distortion Value
0	Name	ASAP
1	Angle Avg Distortion	0.0451805053404261
2	Angle Total Distortion	2966.37125863102
3	Angle Max Distortion	2.5020252671576
4	Area Total Distortion	0.69815188139946
5	Area Max Distortion	0.000410605347394912
6	Edge Total Distortion	0.469720118231062
7	Edge Max Distortion	8.49448183115376E-05
8	L2 Distortion	11438.3325130778
9	L8 Distortion	1959429.82035042
10	QuasiConformal	2.1313838505023
11	DegenerateTriangle	0

LSCM扭曲度量

Index	Distortion Name	Distortion Value
0	Name	LSCM
1	Angle Avg Distortion	0.0452307590636988
2	Angle Total Distortion	2969.67071708621
3	Angle Max Distortion	2.44804893584232
4	Area Total Distortion	0.698419257426683
5	Area Max Distortion	0.00040519810417858
6	Edge Total Distortion	0.468575344962776
7	Edge Max Distortion	8.395983761911E-05
8	L2 Distortion	1242.25246016162
9	L8 Distortion	202867.13697657
10	QuasiConformal	1.39702942513201
11	DegenerateTriangle	0

图 13-4　ASAP 和 LSCM 参数化效果对比

（4）如图 13-5 所示是 ASAP 和 ARAP 的效果对比。从中可以看出，ARAP 能够更好地保持原来三维模型上三角形的面积。例如，兔子头三维模型在 ASAP 参数化后的耳朵部分扭曲很大，而 ARAP 参数化得到的兔子耳朵部分则扭曲较小。在半球三维模型上，ARAP 参数化的结果比 ASAP 参数化结果更近似于一个圆。对于复杂的牛的三维模型，可以看出，ARAP 得到比 ASAP 更好的展开效果。

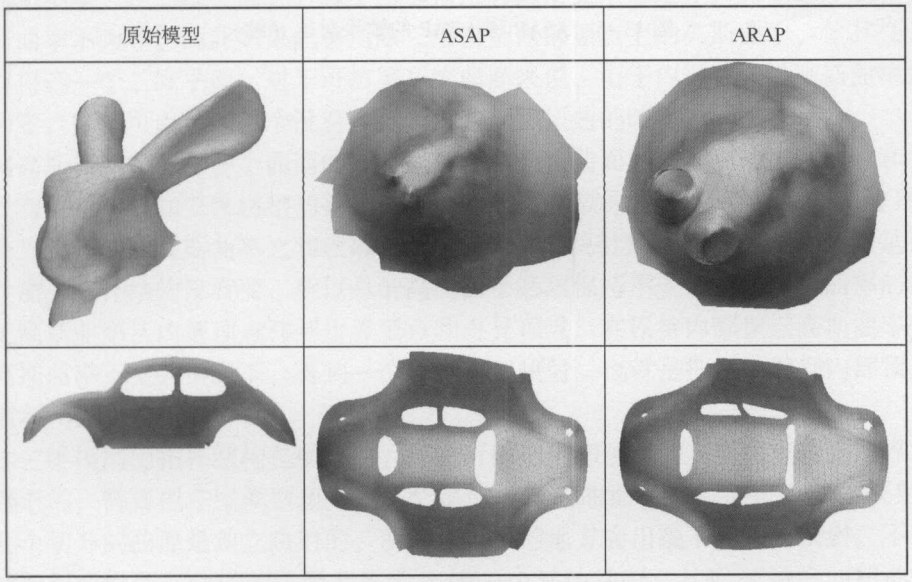


图 13-5　ASAP 和 ARAP 的效果对比

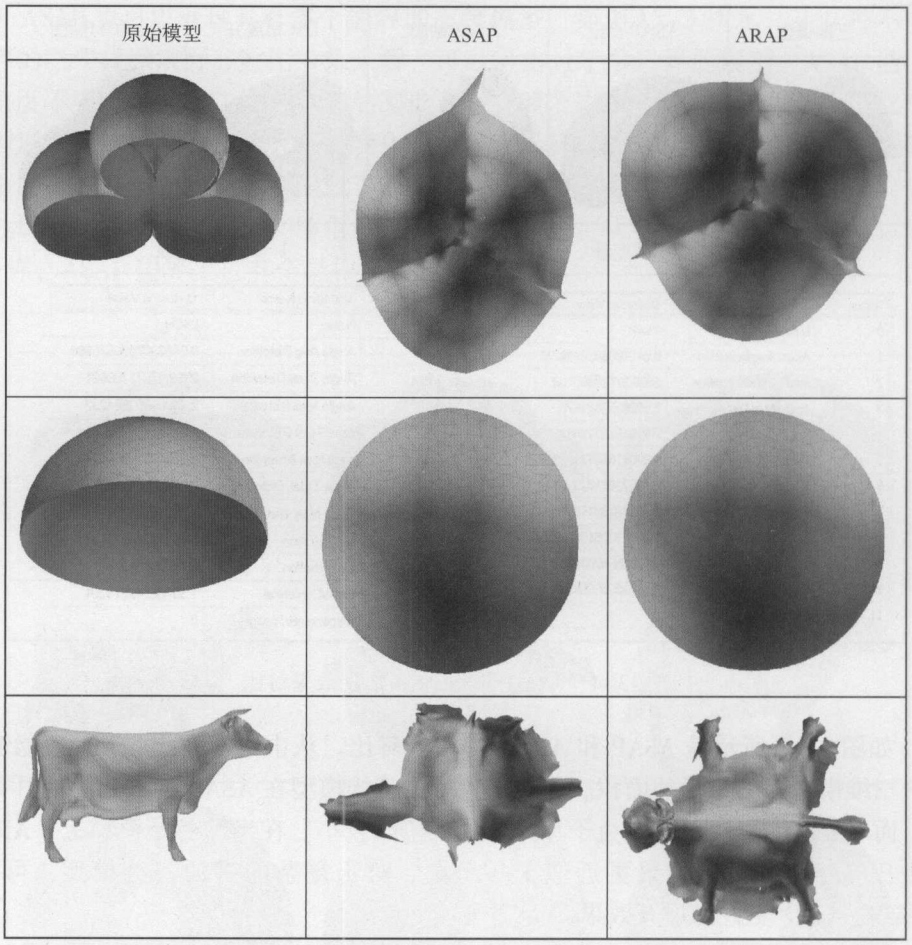


图 13-5　ASAP 和 ARAP 的效果对比（续）

第 14 章 高斯曲率参数化算法

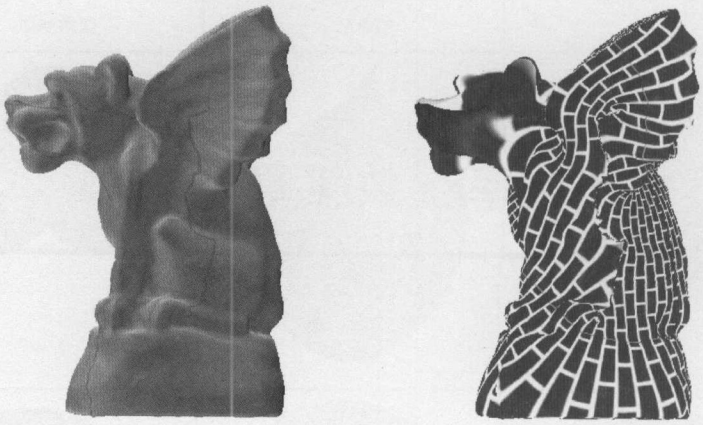
14.1 算法逻辑

三维模型的参数化把三维模型从三维空间转换到一个平面上。在参数化后，三维模型的拓扑结构保持不变，改变的是三维模型的形状，也就是几何信息。从曲率的角度看，三维模型参数化后，每个顶点的曲率发生了变化。每个顶点的曲率是由此顶点的位置和周围顶点的相对位置决定的，因此顶点的位置发生改变，那么曲率就发生变化。前几章讲述的三维模型参数化算法都是直接考虑顶点的位置，然后进行算法的设计的。另一种算法设计思路是采用间接的方法，从顶点的曲率着手，进行算法设计。也就是设计一种算法能够改变三维模型的曲率，从而使三维模型展开到平面上，然后再得到三维模型顶点的位置。基于高斯曲率的参数化算法，通过改变顶点的高斯曲率来把三维模型映射到二维平面。把内部点的高斯曲率移到边界点或奇异点上，在保持总的高斯曲率不变的情况下，使内部点的高斯曲率为零，这种把三维平面展开为二维平面的算法称为高斯曲率参数化算法。

三维模型的参数化希望获得一个二维平面的映射，这个映射要求能够尽可能保持原来的角度和面积不变。在二维平面上，每个内部顶点的高斯曲率都是零，而在三维上，相应的顶点的高斯曲率不为零。因此如果能够消除三维模型内部顶点上的高斯曲率，使其变为零，那么就能够得到一个二维平面。对于可扩展三维曲面来说，由于内部顶点的高斯曲率在三维空间已经为零，因此可以得到一个没有任何角度和面积扭曲的映射。

根据高斯博纳定理，一个曲面的高斯曲率之和是由曲面的拓扑结构决定的，和曲面的几何性质没有关系。假如三维模型的拓扑结构确定了，也就是三维模型的亏格和边界确定了，那么这个三维模型的高斯曲率之和就确定了。三维模型映射到二维平面上改变的是三维模型的几何性质，拓扑结构没有变，所以总的高斯曲率必须保持不变。基于高斯曲率的参数化算法需要把高斯曲率从内部顶点移到边界顶点和奇异顶点。在保持内部顶点高斯曲率为零，以及总的高斯曲率不变的情况下，得到一个二维平面映射。也就是把三维模型内部顶点的高斯曲率数值移动到边界顶点上。

假如三维模型的拓扑结构是碟形的，那么可以把内部顶点的高斯曲率移到边界顶点。假如不是碟形的，需要把三维模型切开为几个碟形拓扑的曲面，然后把每个曲面映射到平面。但因为几个切开的曲面是独立映射的，所以在连接的地方会出现不一致的映射。不同的切割方法造成的扭曲也是不一样的。因此需要选择一个好的切割，从而使最终的扭曲最少。如图 14-1所示，绿色的线条是切割线，三维模型参数化贴图后，切割线的两边的贴图在映射后不一致。

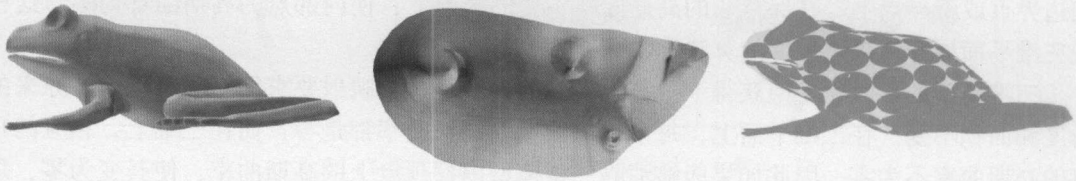


（a）三维模型　　（b）贴图

图 14-1　三维模型剪开贴图

高斯曲率参数化把高斯曲率移到几个奇异顶点上，其他的顶点高斯曲率是零。然后根据当前的高斯曲率，计算三维模型新的度量（Metric），也就是新的边长。得到所有三角形的边长后，就可以把三角形在二维平面上重构。这样的优点是可以先计算边长，再进行切割，从而使得切割开后，连接两个独立曲面的边长映射后是一样的。这个算法的要点是如何找到最优的奇异点，从而使最终的参数化扭曲最小。

高斯参数化实例如图 14-2 所示。



（a）三维模型　　（b）高斯曲率参数化　　（c）贴图

图 14-2　高斯曲率参数化效果

参数化算法的最终目标都是得到三维模型展开后的二维顶点坐标。在线性 ABF 参数化算法中，这个目标分为两步，第一步是得到每个三角形的三个角的角度，然后根据角度得到顶点的二维坐标。高斯曲率参数化最核心的思想是先得到三维模型映射到二维平面上的每个三角形的边长，然后根据边长可以得到每个三角形三个夹角的角度，最后和 ABF 算法一样，根据角度可以得到顶点的位置。因此，高斯曲率参数化的要点就是构建三维模型映射前边长和映射后边长之间的数学关系。三维模型映射前的边长是确定的，三维模型映射后的边长和映射前的边长的关系是受到保角约束限制的。也就是根据映射前的边长和保角约束条件，就可以计算出三维模型映射后的边长。这个映射前后的边长不是直接相关的，而是间接相关的，也就是通过映射前后的高斯曲率来得到的。而映射前后的高斯曲率的数值和两者之间的关系是已经确定的了。

14.2　边长度量

对于光滑的三维曲面来说，一个最重要的概念是度量，也就是度量两点之间的距离。有

了这个度量，各种三维曲面上的属性，如面积等都可以进行计算。在离散的三维模型上，也需要设置一个度量，一个自然的度量就是设置在边上，所有边的边长的集合就是这个三维模型的度量。

第一步：对于一个三维模型 M 来说，离散化的度量（Metric）就是给每个边赋值一个正数，最自然的度量就是这个边的边长。

$$l_{ij} = \| x_i - x_j \| \mid (i,j) \in E \tag{14-1}$$

第二步：根据三角形三个边的边长，可以得到三角形三个角的角度。由这个边长度量得到的角度计算公式为：

$$A_{LM} = \left\{ \begin{matrix} \alpha_v^f = \arccos\left(\dfrac{l_{vu}^2 + l_{vw}^2 + l_{uw}^2}{2 l_{uv} l_{uw}} \right) \\ f = (u,v,w) \in F \end{matrix} \right\} \tag{14-2}$$

第三步：每个顶点的相邻所有角的角度确定后，这个顶点的高斯曲率就确定了。从而由边长度量得到的高斯曲率为：

$$K_{LM} = \left\{ k_v = 2\pi - \sum_{f \in F_v} \alpha_v^f \mid v \in V \right\} \tag{14-3}$$

第四步：如果 $\chi(M)$ 表示欧拉示性数，那么整个模型的高斯曲率满足如下条件：

$$\sum K = 2\pi\chi(M) \tag{14-4}$$

第五步：保角参数化就是把三维曲面映射到二维后，保持角度不变。对于连续曲面来说，可以得到保角映射，但离散曲面由于每个顶点相邻的角度之和不等于 $2\Re$，因此会产生扭曲。基于高斯曲率的保角参数化就是把三维模型的度量保角变换之后得到给定的高斯曲率。

14.3　保角缩放因子

如果把三维模型的每条边作为度量，假如三维模型的映射是保角的，那么三维模型映射前后的度量不是随意的，根据公式具有相应的关系。如果要保持角度不变，那么给每个顶点的度量乘以相应的保角缩放因子得到新的度量，这个度量下的模型和原来的三维模型是保角映射的。如果确定了缩放因子的值，那么边长就确定了。这个保角缩放因子是由高斯曲率决定的。

第一步：对于每个边的边长度量乘以相应的保角缩放因子 $e^{2\phi}$ 后得到新边长，新边长和旧边长度量之间就是保角变换关系。其中缩放因子每个边都可以不一样。

第二步：边长度量在三维模型参数化变换后，会引起相应曲面几何性质改变，也就是高斯曲率会发生变换。变换前后的高斯曲率和保角缩放因子的关系为：

$$\nabla^2 \phi = K^{\text{orig}} - e^{2\phi} K^{\text{new}}$$

第三步：其中 ∇^2 就是拉普拉斯操作符，也就是余切拉普拉斯矩阵，K^{orig} 是原来的高斯曲率，K^{new} 是新的高斯曲率。

第四步：在光滑的情况下，这个方程是非线性的。这是因为在连续的情况下，假如边长度量缩放，那么高斯曲率也会相应的缩放。例如，一个放大的球的高斯曲率比一个小的球的高斯曲率要小。

第五步：而在离散的情况下，放大的四边形和缩小的四边形顶点上的高斯曲率都一样。因此离散情况下上述方程变为线性的关系，也就是：

$$\nabla^2 \phi_v = K_v^{\text{new}} - K_v^{\text{orig}} \tag{14-5}$$

第六步：那么给定了目标高斯曲率 K^{T}，和原来的高斯曲率，那么边长变换缩放因子 e^{ϕ} 可以用如下线性系统得到：

$$\nabla^2 \phi = K^{\text{T}} - K^{\text{orig}} \tag{14-6}$$

第七步：上述方程解出来的缩放因子 ϕ 是定义在每个顶点上的，因此每个边 (i,j) 的缩放因子可以从这条边的两个顶点的缩放因子积分得到：

$$\phi(t) = t\phi_i + (1-t)\phi_j$$

第八步：那么整个边的缩放因子为：

$$S_{ij} = \int_0^1 e^{(t)} \mathrm{d}t = \begin{cases} \dfrac{e^{\phi_j} - e^{\phi_i}}{\phi_j - \phi_i} & \phi_i \neq \phi_j \\ e^{\phi_i} & \phi_i = \phi_j \end{cases} \quad (i,j) \in E \tag{14-7}$$

第九步：在根据高斯曲率得到保角缩放因子之后，就可以计算映射后新的边长为：

$$L^{\text{T}} = \{ l_{ij}^{\text{T}} = l_{ij} \cdot s_{ij} \mid (i,j) \in E \} \tag{14-8}$$

第十步：这个方法计算出来的边长度量是个近似值，因为用这个度量计算高斯曲率和目标高斯曲率有误差。

第十一步：到目前为止，不需要整个模型是拓扑碟形。假如整个模型是碟形的，那么可以把所有的高斯曲率平均分配到边界的顶点上，把内部顶点的高斯曲率设置为零。

第十二步：在得到新的边长后，可以用最小二次方的方法进行重构，得到二维映射后顶点的位置。最小二次方重构也可以把误差平均分布到整个模型，只有在进行重构时才需要把模型切开为碟形拓扑。

14.4 核心代码

高斯曲率参数化算法的关键是确定映射后每个顶点的高斯曲率，假如三维模型是碟形拓扑，那么映射后的高斯曲率可以很容易确定。可以把内部顶点的高斯曲率设置为零，然后把所有的高斯曲率平均分配大边界顶点。高斯曲率参数化的核心函数包括计算所有顶点的高斯曲率，把高斯曲率分配到边界顶点，计算缩放因子，根据缩放因子，计算参数化后的边长，然后根据边长计算角度，最后由角度计算顶点位置这几个函数。

1. 计算三维模型所有顶点的高斯曲率

```
public static double[ ] ComputeGaussianCurvatureIntegrated(TriMesh mesh)
{
    double[ ] gauss = new double[ mesh. Vertices. Count ];

    foreach(var v in mesh. Vertices)
    {
        gauss[ v. Index ] = ComputeGaussianCurvatureIntegrated(v);
```

```
        }
        return gauss;
    }

    public static double ComputeGaussianCurvatureIntegrated(TriMesh.Vertex v)
    {
        bool onBoundary = false;
        double curvature = 0;
        foreach(var hf in v.HalfEdges)
        {
            if(hf.OnBoundary)
            {
                onBoundary = true;
            }
            else
            {
                Vector3D v1 = (hf.Previous.FromVertex.Traits.Position
                               - v.Traits.Position).Normalize();
                Vector3D v2 = (hf.ToVertex.Traits.Position
                               - v.Traits.Position).Normalize();
                curvature -= ComputeAngle(ref v1, ref v2);
            }
        }
        curvature += onBoundary ? Math.PI : 2 * Math.PI;
        return curvature;
    }
```

2. 把三维模型所有的高斯曲率平均转移到边界顶点

```
    private double[] ComputeTargetCurvature(List<TriMesh.Vertex> boundaries,
                                            double[] Guassianes)
    {
        double sumGuassian = 0;
        for(int i = 0; i < Guassianes.Length; i++)
        {
            sumGuassian += Guassianes[i];
        }
        double avgCurvature = sumGuassian/boundaries.Count;

        double[] targetCurvature = new double[Mesh.Vertices.Count];
        for(int i = 0; i < targetCurvature.Length; i++)
        {
```

```
            targetCurvature[i] = 0;
        }
        foreach(TriMesh.Vertex V in boundaries)
        {
            targetCurvature[V.Index] = avgCurvature;
        }
        return targetCurvature;
    }
```

3. 求解线性系统，计算每个顶点的缩放因子

```
public double[] ComputeScaleFactorVertex(double[] target,
                                         double[] origian)
{
    for(int i = 0;i < Mesh.Vertices.Count;i ++)
    {
        origian[i] = target[i] - origian[i];
    }
SparseMatrix Laplace = LaplaceManager.Instance.BuildMatrixRigid(Mesh);
LinearSystem.Instance.Factorize(ref Laplace,EnumSolver.CholmodCholesky);

double[] scaleFactor =
    LinearSystem.Instance.SolveByPreFactorize(ref origian);

LinearSystem.Instance.Free();

return scaleFactor;
}
```

4. 根据每个顶点的缩放因子，计算每个边的缩放因子

```
public double[] ComputeScaleFactorEdge(double[] scaleV)
{
    double[] S = new double[Mesh.Edges.Count];

    foreach(TriMesh.Edge edge in Mesh.Edges)
    {
        TriMesh.Vertex Vi = edge.Vertex0;
        TriMesh.Vertex Vj = edge.Vertex1;

        double value = (Math.Exp(scaleV[Vj.Index])
                        - Math.Exp(scaleV[Vi.Index]))
                       /(scaleV[Vj.Index] - scaleV[Vi.Index]);
```

```
            S[edge.Index] = value;
        }
        return S;
    }
```

5. 根据每条边的缩放因子，计算边长

```
public double[] ComputeNewEdgeLength(double[] scaleEdge)
{
    double[] newLength = new double[Mesh.Edges.Count];

    foreach(TriMesh.Edge edge in Mesh.Edges)
    {
            TriMesh.Vertex Vi = edge.Vertex0;
            TriMesh.Vertex Vj = edge.Vertex1;

        double length = (Vi.Traits.Position
                        - Vj.Traits.Position).Length();

        newLength[edge.Index] = length * scaleEdge[edge.Index];
    }

    return newLength;
}
```

6. 根据边长计算角度

```
private double[] ComputeAngleByEdge(double[] length)
{
    double[] newAngles = new double[Mesh.HalfEdges.Count];

    double angleParameter = 0.01;
    double angle1 = (Math.PI/180) * angleParameter;
    double angle2 = (Math.PI/180) * (180 - 2 * angleParameter);

    foreach(TriMesh.Face face in Mesh.Faces)
    {
        foreach(TriMesh.HalfEdge hf in face.Halfedges)
        {
            double next = length[hf.Next.Edge.Index];
            double pre = length[hf.Previous.Edge.Index];
            double current = length[hf.Edge.Index];
```

```
                double cosA = ((current * current
                                + next * next - pre * pre)
                               /(2 * current * next));

                if(cosA > 1)
                {
                    cosA = Math. Cos(angle1);
                }
                else if(cosA < -1)
                {
                    cosA = Math. Cos(angle2);
                }
                double angle = Math. Acos(cosA);
                newAngles[hf. Next. Index] = angle;
            }
        }
        return newAngles;
    }
```

7. 根据角度计算顶点的位置

```
    private void Reconstruction(double[] newAngles)
    {
        ParameterizationReconstruction flat
            = new ParameterizationReconstruction(Mesh);

        int[] hfIndices = new int[Mesh. HalfEdges. Count];
        for(int i = 0;i < hfIndices. Length;i ++)
        {
            hfIndices[i] = i;
        }

        flat. HfOld2NewIndex = hfIndices;
        flat. LeastSquareReconstruction(newAngles,
                        out UnknownU,out UnknownV);
    }
```

8. 整个参数化过程

```
    public override void Parameterize()
    {
        TriMesh clone = TriMeshIO. Clone(Mesh);

        List < TriMesh. Vertex > all = RetrieveSingularity();
```

```
        double[ ] Guassianes =
            TriMeshUtil. ComputeGaussianCurvatureIntegrated( Mesh) ;

        double[ ] targetCurvature = ComputeTargetCurvature( all, Guassianes) ;

        double[ ] scaleFactorVertex =
            ComputeScaleFactorVertex( targetCurvature, Guassianes) ;

        double[ ] scaleFactorEdge = ComputeScaleFactorEdge( scaleFactorVertex) ;
        double[ ] length = ComputeNewEdgeLength( scaleFactorEdge) ;


        double[ ] newAngles = ComputeAngleByEdge( length) ;

        Reconstruction( newAngles) ;

    }
```

14.5　实验分析

高斯曲率参数化算法采用了和之前几章介绍的参数化算法截然不同的算法设计思路。高斯曲率参数化算法核心思路是改变三维模型的曲率，从而间接改变三维模型的顶点。三维模型顶点的曲率是已知的，映射到平面后的顶点曲率也是可以设定的。从而根据两个曲率的关系，间接求得顶点映射后的位置。高斯曲率的曲率改变不是任意的，是受到高斯博纳定理约束的。需要保持整个三维模型高斯曲率之和在映射前和映射后不发生变化。最后可以构造一个线性系统来求解。从高斯曲率参数化算法中可以看出，针对一个具体的问题，有时可以通过间接的方法来得到结果。在高斯曲率参数化设计中，可以总结出对于一个问题，如果输入和输出不是线性的关系，可以采用如下几种方法把线性的关系变为非线性关系。第一种方法是把非线性系统简化为线性系统，从而得到线性的关系。例如，在线性 ABF 算法中所采用的利用泰勒展开，把非线性的约束近似为线性的约束。第二种方法是通过把输入和输出非线性的变换转换到其他空间，然后在其他空间构造一个线性关系。这样虽然输入和输出本质上还是非线性的，但核心计算部分变为线性的。第三种方法是采用间接的方法，也就是不直接求解输出，而是求解一个中间变量，这个中间变量和输入是线性的关系。然后根据中间变量再用非线性的计算求得最终的输出。

（1）高斯参数化由于把所有的高斯曲率平均分配到每个顶点，因此边界上每个顶点的高斯曲率都近似一样，从而构成一个圆形的边界。但这与和谐参数化固定边界的算法不一样。边界顶点的位置不是预先固定的，预先固定的是边界顶点的高斯曲率。如图 14-3 所示，可以看出三维模型剪开后的边界都是尽可能近似于一个圆形。

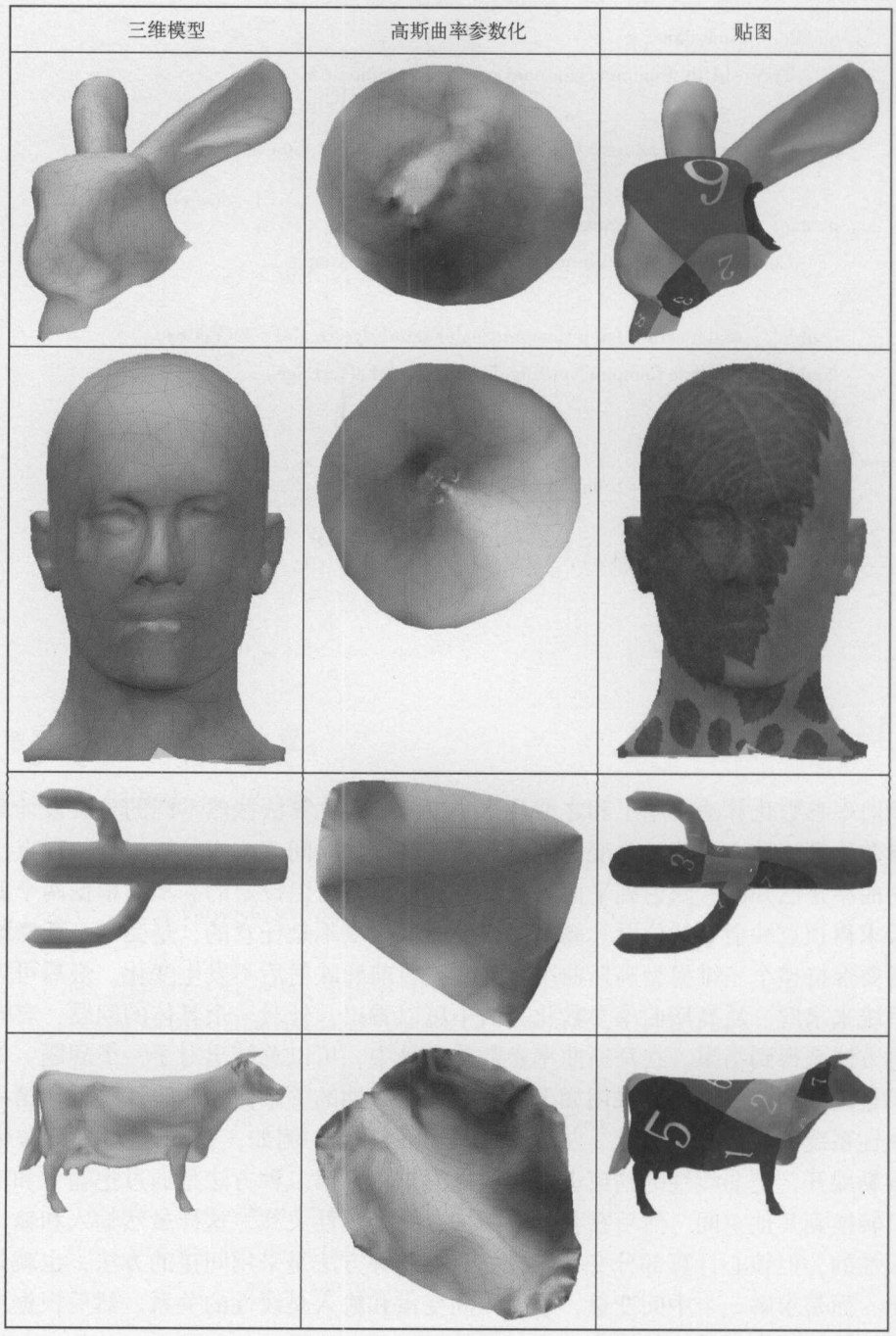


图 14-3　高斯曲率参数化效果图

（2）如果三维模型具有多个边界，高斯曲率算法把所有的高斯曲率平均分配到每个边界顶点，没有对顶点所属的边界进行区分，因此在三维模型展开后会发生重叠覆盖现象。这是由于高斯曲率算法的设计无法对不同的边界进行分别处理，如图 14-4 所示。

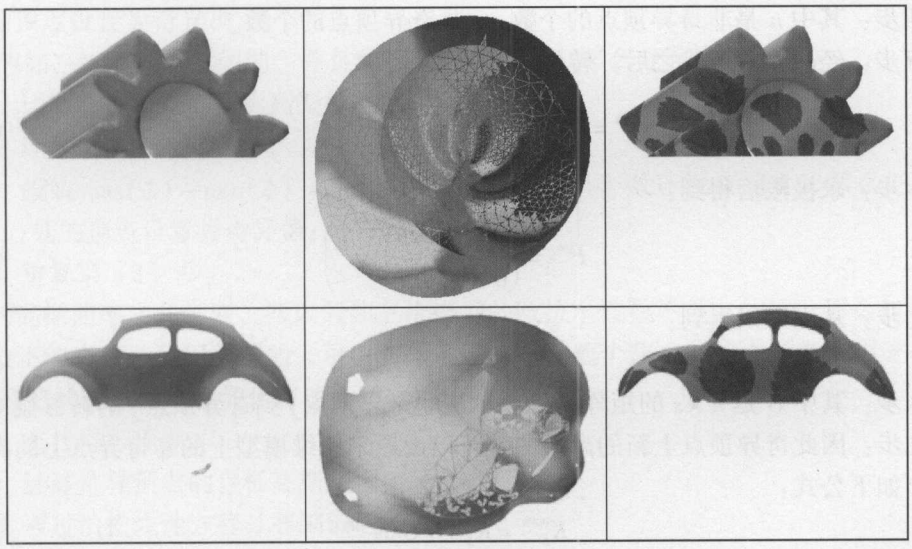


图 14-4　多个边界三维模型高斯曲率参数化效果图

14.6　奇异顶点

对于碟形拓扑的三维模型，可以简单地把所有的高斯曲率移动到边界顶点。但对于其他拓扑的三维模型，需要把三维模型剪开，那么找到一些特殊的顶点，从而沿着这些顶点剪开能够得到最优化的结果。

1. 奇异顶点的概念

奇异顶点（Cone Singularity）是指展开后高斯曲率不为零的特殊顶点。为了使其他顶点的高斯曲率为零，需要把这些顶点的高斯曲率移动到奇异顶点。三维模型上奇异顶点的查找是一个迭代的过程。每次迭代每个不是奇异的顶点把自己的高斯曲率平均散发给邻居顶点，从而使顶点的高斯曲率进行扩散，而每个奇异顶点不散发高斯曲率，而是尽可能地吸收高斯曲率。假如所有的高斯曲率都被奇异顶点吸收，那么迭代终止。在整个迭代过程中，因为没有高斯曲率增加和减少，所以高斯曲率之和保持不变。这个迭代过程可以看作一个吸收**马尔科夫链**。

2. 马尔可夫链迭代过程

第一步：每个顶点是一个状态，从一个顶点到另一个顶点的转移概率为：

$$P_{ij}=\begin{cases} w_{ij} & (i,j)\in E, i\notin S, \sum_j w_{ij}=1 \\ 1 & i=j, i\in S \\ 0 & \text{其他} \end{cases} \tag{14-9}$$

第二步：在这个马尔科夫链中，如果漫游到一个非奇异顶点，那么继续漫游，而一个奇异顶点是一个吸收状态，漫游到达奇异顶点后就停止。

第三步：把所有的奇异顶点排序到最后，那么可以得到如下的状态转移矩阵：

$$P=\begin{pmatrix} S_{nxn} & T_{nxs} \\ 0_{sxn} & I_{sxs} \end{pmatrix} \tag{14-10}$$

第四步：其中 n 是非奇异顶点的个数，s 是奇异顶点的个数。

第五步：经过 k 次迭代之后，转移矩阵为：

$$P^k = \begin{pmatrix} S^k & T(I+S+\cdots+S^{k-1}) \\ 0 & I \end{pmatrix} \tag{14-11}$$

第六步：求极限后得到：

$$P^\infty = \begin{pmatrix} 0 & (I-S)^{-1}T \\ 0 & I \end{pmatrix} \tag{14-12}$$

第七步：从而可以得到：

$$G = (I-S)^{-1}T \tag{14-13}$$

第八步：其中 G 是 $n \times s$ 的矩阵，G_{ij} 表示从非奇异顶点 j 到奇异顶点 j 的转移概率。

第九步：因此奇异顶点上新的高斯曲率可以由原来三维模型上的非奇异点上的高斯曲率计算得出如下公式：

$$K_S^{\text{New}} = K_S^{\text{orig}} + G^{\text{T}} K_{V\setminus S}^{\text{orig}} \tag{14-14}$$

第十步：因为矩阵 P 和相邻矩阵的结构类似，因此计算矩阵 G 的每一列向量的方法分为两步。

第十一步：首先根据

$$L\hat{G}_i = \delta_i \tag{14-15}$$

其中 δ_i 是一个长度为 $V = n + s$ 的列向量，这个列向量中第 i 行是 1，其余是 0。

第十二步：计算出长度为 $V = n + s$ 的列向量 $\hat{G}_i$，然后 G 的一个长度为 n 的列向量 G_i 是长度为 $n + s$ 的列向量 $\hat{G}_i$ 的子向量，从 $\hat{G}_i$ 中可以得到 G_i。

第十三步：那么每个 G_i 都可以看作非奇异顶点上的一个函数，也就是离散格林函数。在每个非奇异顶点上，所有列向量 G_i 加起来为 1。因为每个非奇异顶点上的高斯曲率最终分配到奇异顶点上。

寻找奇异顶点问题是指给定一个三维模型和一个二维映射，那么需要寻找奇异顶点，从而使映射的扭曲最小。这是一个“先有鸡还是先有蛋”的问题，因为奇异顶点就是用来进行映射的。这样的问题通常可以用迭代的方法来解决。

3. 奇异值的位置

离散的情况下，假如所有的边长度量的缩放因子都是一样的，也就是都是 e^{ϕ}，那么 ϕ 加减一个常数得到的高斯曲率不变。因此可以假设 ϕ 的平均值是零。假如 ϕ 都是零，那么边长度量不变，高斯曲率不变，因此扭曲是零。可扩展的曲面情况下，可以出现这种情况。通常来说，最大的扭曲出现在 ϕ 值最大和最小的地方。因此，把奇异值放在扭曲最大的地方，可以使周围的扭曲变小。这种奇异值的选择和把高斯曲率绝对值最大的顶点作为奇异值的效果不同，因为离散的高斯曲率是一个局部属性，只和一个顶点的邻居顶点有关。而 ϕ 是由全局范围内计算出来的。

4. 奇异值查找算法

（1）初始化奇异值。

① 假如三维模型有边界顶点，那么所有边界顶点都是奇异值；

② 假如三维模型是封闭的，并且欧拉示性数是正值，那么把高斯曲率为最大正值和最

小负值的顶点设置为奇异顶点。

③ 假如三维模型是封闭的，并且欧拉示性数是零，那么奇异值集合为空。

（2）计算奇异顶点的目标高斯曲率。

（3）计算出 ϕ。

（4）假如 $\max(\phi)-\min(\phi)>\varepsilon$，或者设定的奇异值个数还没有满足，那么把 $\max(\phi)$ 和 $\min(\phi)$ 处的顶点设置为奇异值；

（5）重复第（2）步。

因为高斯曲率之和不变，所以假如三维模型的欧拉示性数不是 1 的话，那么初始化的奇异值个数不能为零。使用不同的 ε 可以对应不同的奇异值个数，试验中通常采用 $\varepsilon=1$。

5. 完整的高斯曲率参数化算法

（1）查找奇异顶点。

（2）计算奇异顶点的目标高斯曲率。

（3）通过泊松线性方程计算缩放因子 ϕ。

（4）通过 ϕ 计算映射到二维的边长。

（5）由新的边长计算出三角形的角度。

（6）根据三角形的角度，用最小二次方法计算最终顶点的映射位置。

在第（3）步求解线性系统时，采用余切权重 $w_{ij}=0.5(\cot(\alpha)+\cot(\beta))$。这个线性系统的矩阵的零度（Co－Rank）是 1。因此，计算的结果 ϕ 加减上一个常量都满足原来的线性系统。这和整个三维模型缩放后，高斯曲率不变是一致的。在上述的线性系统方程中，左边的拉普拉斯矩阵每个列向量之和为 0，而因为高斯曲率之和不变，所以方程组右边列向量之和为 0。因此线性系统存在解。在马尔科夫链里，矩阵 P 是 $P_{ij}=w_{ij}/\sum_j w_{ij}$。直接计算 $L=I-S$ 的逆矩阵是不可行的，因此可以通过求解线性方程来间接计算。计算的次数和奇异顶点个数一样。因为计算出来的边长不一定满足三角形两边之和大于第三边的定律，所以从边长计算角度时可能出现复数，只取复数的真值部分作为角度。

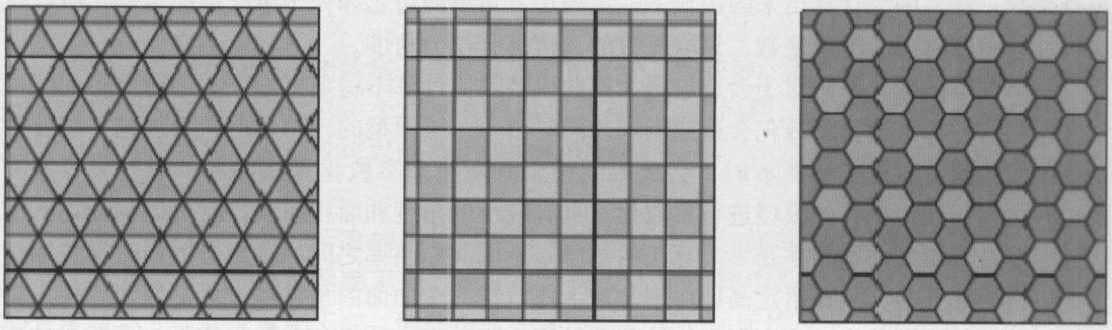
高斯曲率参数化算法对于会造成很大扭曲模型数值计算不稳定，如果不进行更多的切割而直接用这种算法进行参数化会造成困难。同时由于三维模型的三角形形状不规则，拉普拉斯矩阵有负值，因此在计算 ϕ 时，会造成问题。高斯曲率参数化算法简单、有效、容易实现，并可以对任何拓扑的模型进行参数化。具有较小的角度和面积扭曲，且可以保证切开的边界上非奇异点的高斯曲率是零。算法的基础是不同高斯曲率之间的度量可以通过保角缩放因子进行局部变换，从而给定高斯曲率之后，可以把一个曲面的顶点经过局部保角变换后得到目标曲面的高斯曲率。这个保角缩放因子被用来判断奇异顶点的位置。这是一个通过曲率定制和度量缩放的保角参数化算法。

之前讲述的三维模型参数化算法只能作用在碟形的三维模型上，这是由二维平面的拓扑本质所限制的。也就是所有的参数化算法都只能作用在碟形的三维模型上。非碟形的三维模型只有剪开为碟形的拓扑结果才能进行参数化。高斯曲率参数化算法可以先计算边长，然后再切割，因为切割后的两个边边长一样，从而可以在切割线的两边展平后可以得到比较一致的效果。

第 15 章 重新网格化

15.1 介绍

通过软件建模，或者扫描出来得到的三维模型常常质量不好。也就是三角形的大小不均匀，或者很多三角形有钝角。这样的三维模型在很多三维模型处理的算法上得到失败或不好的结果。因此在运行三维模型处理算法时，有时需要对原来的三维模型进行预处理，对原来的三维模型进行优化。有一种优化方法是从原来的三维模型生成新的三维模型，但保持原来的形状不变。新的三维模型具有不同的顶点数量和面的数量，但每个面具有更好的形状，并且面的大小都一致。这种优化的方法称为三维模型的重新网格化。除了三角形面三维模型外，重新网格化还可以生成四边形面、六边形面的三维模型。重新网格化的算法首先将三维模型用参数化算法映射到平面，然后用正三角形、正四边形、正六边形填充这个平面，最后把这些正三角形、正四边形、正六边形的顶点根据三维模型参数化的结果再映射到三维模型上。用大小一样的正多边形填充一个平面只有三种方式：正三角形、正四边形和正六边形，如图 15-1 所示。



（a）正三角形　　（b）正四边形　　（c）正六边形

图 15-1　平面正多边形填充

如图 15-2 和图 15-3 所示是两个三维模型进行重新网格化后得到的三角形面、四边形面和六边形面的新的三维模型。从中可以看出，新的三维模型上的面质量更好，在三角形面上，几乎都是正三角形，并且各个三角形大小都差不多一样。四边形和六边形的情况也类似。

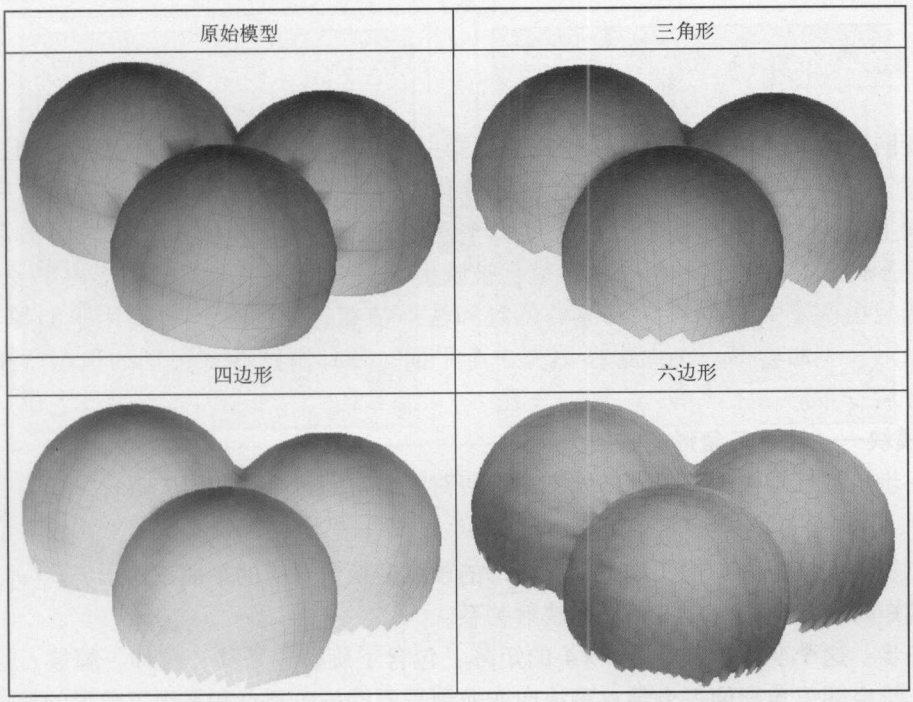


图 15-2　3 个半球三维模型的重新网格化

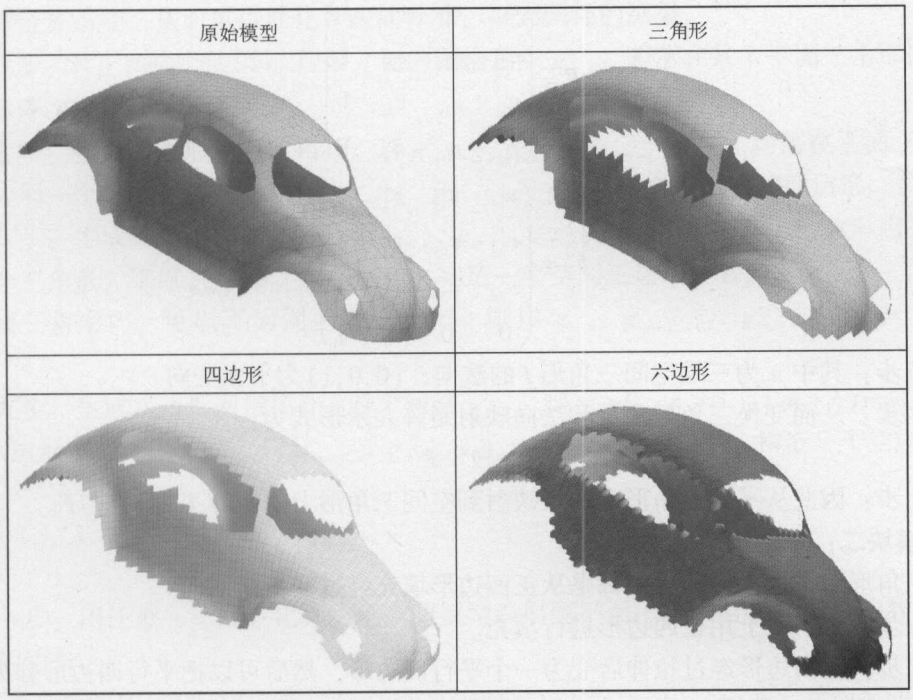


图 15-3　汽车三维模型的重新网格化

15.2 算法步骤

重新网格化的算法可以分为两个模块，第一个模块是确定平面上任意一点到三维模型上的映射。这一个模块是由三维模型的参数化结果决定的。三维模型参数化之后，原来三维空间上的每个三角形就和平面上的每个三角形一一对应，具有一个映射关系，那么平面上位于此三角形内部的每个点，就会被映射到对应的三维空间三角形内部的一个点。映射的函数由两个对应的三角形顶点位置和法向决定。第二个模块是用平面填充算法，用正三角形、正四边形、正六边形填充一个平面，并计算这些多边形的顶点所在的三维模型展开后三角形。

1. 模块一：计算三角形的映射关系

第一步：把三维模型映射到二维平面上，对三维空间每个三角形 $f=(v_1,v_2,v_3)$，存在变换矩阵 T。

第二步：变换矩阵 T 把三维空间三角形的顶点变换为对应的平面三角形 $f'=(v_1',v_2',v_3')$。这个变换矩阵就是两个对应三角形的映射关系。

第三步：这个变换矩阵是个 4×4 的矩阵，包含了旋转、平移、拉伸、缩放。这个变换矩阵把三维空间三角形的每个顶点和法向变换到平面的相应顶点和平面三角形的法向。

第四步：因此从两个对应的三角形的顶点和法向可以构建两个 4×4 的矩阵，如下所示。

$$F=\begin{pmatrix} v_{1x} & v_{1y} & v_{1z} & 1 \\ v_{2x} & v_{2y} & v_{2z} & 1 \\ v_{3x} & v_{3y} & v_{3z} & 1 \\ n_x & n_y & n_z & 1 \end{pmatrix}$$

$$F'=\begin{pmatrix} v'_{1x} & v'_{1y} & v'_{1z} & 1 \\ v'_{2x} & v'_{2y} & v'_{2z} & 1 \\ v'_{3x} & v'_{3y} & v'_{3z} & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

第五步：其中 n 为三维空间三角形 f 的法向，$(0,0,1)$ 为平面法向。

第六步：从而变换三角形顶点和法向映射矩阵表示形式为：

$$FT=F'$$

第七步：因此从平面三角形上的点映射到空间三角形上的矩阵 $T^{-1}=F'^{-1}F$。

2. 模块二：正多边形填充平面

正三角形和正六边形的填充都是从正四边形填充经过变换得到的。

第一步：在平面上用正四边形进行填充。

第二步：正四边形经过拉伸后变为一个平行四边形，然后可以把平行四边形分为两个正三角形，如图 15-4 所示。

第三步：经过第二步得到的正三角形填充中，可以把 6 个正三角形组合得到一个正六边形。

第四步：正四边形进行拉伸变换的公式如下。

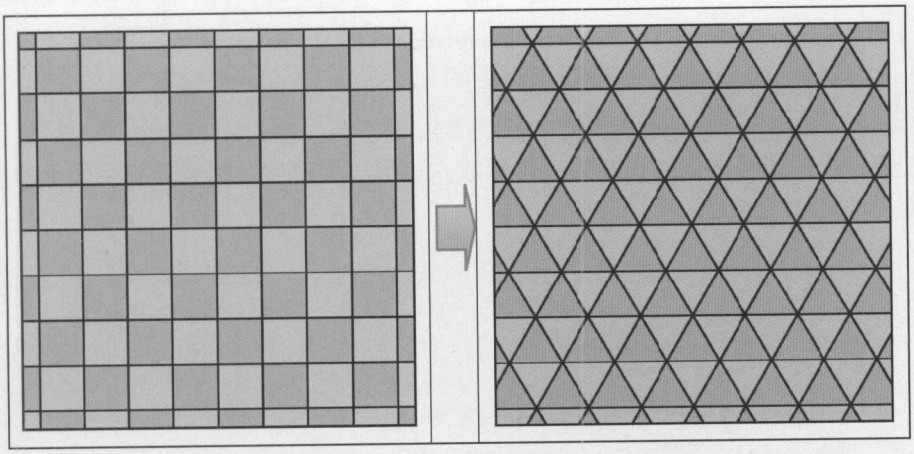


图 15-4　平行四边形

$$\begin{cases} x' = x + \dfrac{y}{2} \\ y' = \dfrac{2}{\sqrt{5}} y \end{cases}$$

第五步：在两个三角形进行映射时，需要计算一个变换矩阵，因此可以把正四边形的拉伸变换合并到变换矩阵中。这样可以对参数化展平后的三角形进行拉伸，即可保持原来正四边形顶点位置不变，从而可以简化算法的计算，得到同样的结果。

第六步：对正四边形的第 m 行第 n 列的网格顶点 $g'_{m,n}$，若落在某个平面三角形内，则右乘变换矩阵 T^{-1} 得到空间坐标 $g_{m,n}$。

第七步：生成四边形网格的拓扑。若 $g'_{m,n}$，$g'_{m+1,n}$，$g'_{m+1,n+1}$，$g'_{m,n+1}$ 均落在平面三角形内，可以不为同一个三角形。则以 $g_{m,n}$，$g_{m+1,n}$，$g_{m+1,n+1}$，$g_{m,n+1}$ 构造一个空间四边形。

第八步：生成三角形网格拓扑。若 $g'_{m,n}$，$g'_{m+1,n}$，$g'_{m+1,n+1}$ 均落在平面三角形内，可以不为同一个三角形，则以 $g_{m,n}$，$g_{m+1,n}$，$g_{m+1,n+1}$ 构造一个空间三边形；若 $g'_{m,n}$，$g'_{m+1,n+1}$，$g'_{m,n+1}$ 均落在平面三角形内，可以不为同一个三角形，则以 $g_{m,n}$，$g_{m+1,n+1}$，$g_{m,n+1}$ 构造一个空间三边形。

第九步：生成六边形网格拓扑。对于网格 $g'_{m,n}$，则 $m>0$、$n>0$ 且 $(m+n)\bmod 3=0$，则 $g_{m,n}$ 为六边形中心，以 $g_{m-1,n-1}$，$g_{m-1,n}$，$g_{m,n+1}$，$g_{m+1,n+1}$，$g_{m+1,n}$，$g_{m,n-1}$ 构造一个空间六边形。

15.3　实现代码

第一步：用任意一种参数化算法把空间模型映射到平面上，同时根据网格的数量计算正四边形的边长。

```
public Remeshing(TriMesh mesh,int num)
{
    this. mesh = mesh;
    this. gridSize = 1. 0/num;
```

```
        ParameterizationARAP arap = new ParameterizationARAP(mesh);
        uvArr = arap.ParameterizeUV(mesh);
    }
```

第二步：若重新网格化的目标为三角形或六边形，则对参数化展开到平面上的模型进行拉伸变换。

```
    private void Shear()
    {
        for(int i = 0;i < uvArr.Length;i ++)
        {
            uvArr[i].x + = uvArr[i].y/2d;
            uvArr[i].y * = 2d/Math.Sqrt(5);
        }
    }
```

第三步：用正四边形填充大小为 1 × 1 的平面。

```
    private void CreateGrid()
    {
        Vector2D min;
        Vector2D max;
        VectorUtil.GetRect(this.uvArr,out min,out max);
        this.grid = new PlanarGrid(min,this.gridSize,this.gridSize);
    }
    public class PlanarGrid
    {
        Vector2D min;
        double stepM;
        double stepN;

        public Vector2D this[PlanarGridCell cell]
        {
            get {return this[cell.m,cell.n];}
        }

        public Vector2D this[int m,int n]
        {
            get {return min + new Vector2D(m * stepM,n * stepN);}
        }

        public PlanarGrid(Vector2D min,double stepM,double stepN)
        {
            this.min = min;
```

```
        this.stepM = stepM;
        this.stepN = stepN;
    }

    public PlanarGrid(Vector2D min,Vector2D max,int m,int n)
    {
        this.min = min;
        this.stepM = (max.x - min.x)/m;
        this.stepN = (max.y - min.y)/n;
    }

    public PlanarGridCell Floor(Vector2D p)
    {
        int m = (int)Math.Floor((p.x - min.x)/stepM);
        int n = (int)Math.Floor((p.y - min.y)/stepN);
        return new PlanarGridCell(m,n);
    }

    public PlanarGridCell Ceiling(Vector2D p)
    {
        int m = (int)Math.Ceiling((p.x - min.x)/stepM);
        int n = (int)Math.Ceiling((p.y - min.y)/stepN);
        return new PlanarGridCell(m,n);
    }
}
```

第四步：计算从平面三角形到空间三角形的变换矩阵。

```
public Matrix4D ComputeTransform(Vector3D faceNormal,
                    Vector3D[] pos,Vector2D[] uv)
{
    Matrix4D from = new Matrix4D();
    from.Row1 = new Vector4D(uv[0],0,1);
    from.Row2 = new Vector4D(uv[1],0,1);
    from.Row3 = new Vector4D(uv[2],0,1);
    from.Row4 = new Vector4D(0,0,1,1);

    Matrix4D to = new Matrix4D();
    to.Row1 = new Vector4D(pos[0],1);
    to.Row2 = new Vector4D(pos[1],1);
    to.Row3 = new Vector4D(pos[2],1);
    to.Row4 = new Vector4D(faceNormal,1);
```

```
        Matrix4D m = from. Inverse( ) * to;
        return m;
    }
```

第五步：计算平面三角形的最小坐标和最大坐标，得到矩形范围。

```
    public PlanarTriangle(params Vector2D[ ] points)
    {
        if(points. Length ! =3)
        {
            throw new Exception("点数不为3");
        }
        for(int i =0;i <3;i ++)
        {
            lines[i] = new DirLine(points[i],points[(i+1)%3]);
        }
        VectorUtil. GetRect(points,out this. min,out this. max);
    }
```

第六步：获得矩形范围内的所有正四边形的顶点，遍历判断每个顶点是否位于平面三角形内部。使用分治法避免直接遍历网格全部顶点。

```
    public PlanarGridCell[ ] Contains(PlanarGrid grid)
    {
        List < PlanarGridCell > list = new List < PlanarGridCell > ( );
        PlanarGridCell minCell = grid. Ceiling(this. min);
        PlanarGridCell maxCell = grid. Floor(this. max);
        for(int i = minCell. m;i < = maxCell. m;i ++)
        {
            for(int j = minCell. n;j < = maxCell. n;j ++)
            {
                if(this. Contains(grid[i,j]))
                {
                    list. Add(new PlanarGridCell(i,j));
                }
            }
        }
        return list. ToArray( );
    }
    public bool Contains(Vector2D p)
    {
        foreach(var line in this. lines)
        {
            if(line. Intersect(p) == DirLineIntersection. Right)
```

```
            }
                return false;
            }
        }
        return true;
    }
```

第七步：计算位于此平面三角形内部所有正四边形顶点所对应的三维空间坐标。

```
    private Matrix4D MapPoint(PlanarGridCell[ ] cells,Matrix4D m)
    {
        foreach(var cell in cells)
        {
            Vector3D v2 = new Vector3D(grid[cell],0);
            Vector3D v3 = TriMeshUtil. TransformationVector3D(v2,m);
            this. dict[cell] = new PV( ) {Pos = v3};
        }
        return m;
    }
```

第八步：对于三维模型参数化后的每个三角形面，把位于每个面内部的正四边形顶点坐标映射到三维空间。

```
    public void Map( )
    {
        this. dict = new Dictionary < PlanarGridCell,PV > ( );
        foreach(var face in this. mesh. Faces)
        {
            Vector3D[ ] pos = TriMeshUtil. GetVertices(face,p => p. Traits. Position);
            Vector2D[ ] uv = TriMeshUtil. GetVertices(face,p => this. uvArr[p. Index]);
            PlanarTriangle t = new PlanarTriangle(uv);
            PlanarGridCell[ ] cells = t. Contains(this. grid);
            Matrix4D m = ComputeTransform(face. Traits. Normal,pos,uv);
            m = MapPoint(cells,m);
        }
    }
```

第九步：构建三角形面三维模型，遍历所有映射后的顶点，添加三维模型的顶点。

```
    private TriMesh CreateVertexTriangle( )
    {
        TriMesh tri = new TriMesh( );
        foreach(var item in this. dict)
        {
            HalfEdgeMesh. Vertex v =
```

```
                tri.Vertices.Add(new VertexTraits(item.Value.Pos));
            item.Value.Vertex = v;
        }
        return tri;
    }
```

第十步：构建三角形面三维模型的拓扑，给三维模型添加三角形面。

```
    private void CreateTopologyTriangle(TriMesh tri)
    {
        foreach(var item in this.dict)
        {
            PlanarGridCell cell = item.Key;
            PlanarGridCell right = new PlanarGridCell(cell.m + 1,cell.n);
            PlanarGridCell bottom = new PlanarGridCell(cell.m,cell.n + 1);
            PlanarGridCell rb = new PlanarGridCell(cell.m + 1,cell.n + 1);
            if(this.dict.ContainsKey(right)&& this.dict.ContainsKey(rb))
            {
                tri.Faces.AddTriangles(this.dict[cell].Vertex,
                    this.dict[right].Vertex,this.dict[rb].Vertex);
                if(dict.ContainsKey(bottom))
                {
                    tri.Faces.AddTriangles(this.dict[cell].Vertex,
                        this.dict[rb].Vertex,this.dict[bottom].Vertex);
                }
            }
        }
    }
```

第十一步：三角形面三维模型重新网格化完整函数。

```
    public TriMesh BuildTriangle()
    {
        Shear();
        this.CreateGrid();
        this.Map();
        TriMesh tri = CreateVertexTriangle();
        CreateTopologyTriangle(tri);
        return tri;
    }
```

第十二步：生成四边形三维模型，添加三维模型顶点。

```
    private QuadMesh CreateVertexQuad()
    {
```

```
        QuadMesh quad = new QuadMesh();
        foreach(var item in this.dict)
        {
            HalfEdgeMesh.Vertex v =
                quad.Vertices.Add(new VertexTraits(item.Value.Pos));
            item.Value.Vertex = v;
        }
        return quad;
    }
```

第十三步：构建四边形三维模型拓扑，添加三维模型四边形面。

```
    private void CreateTopologyQuad(QuadMesh quad)
    {
        foreach(var item in this.dict)
        {
            PlanarGridCell cell = item.Key;
            PlanarGridCell right = new PlanarGridCell(cell.m + 1,cell.n);
            PlanarGridCell bottom = new PlanarGridCell(cell.m,cell.n + 1);
            PlanarGridCell rb = new PlanarGridCell(cell.m + 1,cell.n + 1);
            if(this.dict.ContainsKey(right)&&
               this.dict.ContainsKey(bottom)&&
               this.dict.ContainsKey(rb))
            {
            quad.Faces.AddQuads(this.dict[cell].Vertex,this.dict[right].Vertex,
                this.dict[rb].Vertex,this.dict[bottom].Vertex);
            }
        }
    }
```

第十四步：四边形面三维模型重新网格化完整函数。

```
    public QuadMesh BuildQuad()
    {
        this.CreateGrid();
        this.Map();
        QuadMesh quad = CreateVertexQuad();
        CreateTopologyQuad(quad);
        return quad;
    }
```

第十五步：生成六边形三维模型，添加三维模型顶点。

```
    private QuadMesh CreateVertexHex()
    {
```

```
        QuadMesh hex = new QuadMesh();
        foreach(var item in this.dict)
        {
            int m = item.Key.m;
            int n = item.Key.n;
            if(m>0 && n>0 &&(m+n)%3 !=0)
            {
                HalfEdgeMesh.Vertex v =
                    hex.Vertices.Add(new VertexTraits(item.Value.Pos));
                item.Value.Vertex = v;
            }
        }
        return hex;
    }
```

第十六步：构建六边形三维模型拓扑，添加三维模型六边形面。

```
    private void CreateTopologyHex(QuadMesh hex)
    {
        foreach(var item in this.dict)
        {
            if(item.Value.Vertex == null)
            {
                int m = item.Key.m;
                int n = item.Key.n;
                PlanarGridCell[] round = new PlanarGridCell[6]{
                    new PlanarGridCell(m-1,n-1),
                    new PlanarGridCell(m,n-1),
                    new PlanarGridCell(m+1,n),
                    new PlanarGridCell(m+1,n+1),
                    new PlanarGridCell(m,n+1),
                    new PlanarGridCell(m-1,n)
                };
                HalfEdgeMesh.Vertex[] arr = new HalfEdgeMesh.Vertex[6];
                bool contain = true;
                for(int i=0;i<6;i++)
                {
                    contain = this.dict.ContainsKey(round[i]);
                    if(contain)
                    {
                        arr[i] = this.dict[round[i]].Vertex;
                    }
                    else
```

```
                    {
                        break;
                    }
                }
                if(contain)
                {
                    try
                    {
                        hex. Faces. Add(arr);
                    }
                    catch(Exception e)
                    {
                        Console. WriteLine(e. Message);
                    }
                }
            }
        }
    }
```

第十七步：六边形面三维模型重新网格化完整函数。

```
public QuadMesh BuildHex()
{
    Shear();
    this. CreateGrid();
    this. Map();
    QuadMesh hex = CreateVertexHex();
    CreateTopologyHex(hex);
    this. ComputeHexNormal(hex);
    return hex;
}
```

第十八步：平面三角形，由三条有向线组成。根据平面有向线，可以判断点和线的关系。

```
public enum DirLineIntersection
{
    Intersecting,
    Left,
    Right
}
public struct DirLine
{
    public static double ZeroTolerance = 1e - 6d;
```

```
        public double A;
        public double B;
        public double C;

        public DirLine(Vector2D from,Vector2D to)
        {
            A = to.y - from.y;
            B = from.x - to.x;
            C = from.y * to.x - from.x * to.y;
        }

        public DirLineIntersection Intersect(Vector2D p)
        {
            double d = A * p.x + B * p.y + C;
            if(d <- ZeroTolerance)
            {
                return DirLineIntersection.Left;
            }
            else if(d > ZeroTolerance)
            {
                return DirLineIntersection.Right;
            }
            else
            {
                return DirLineIntersection.Intersecting;
            }
        }
    }
```

15.4 实验效果

1. 三角形重新网格化

如图 15-5 所示是对各种三维模型进行三角形重新网格化的结果。从中可以看出，经过参数化后展开为平面，然后重新采样得到的三维模型仍然能够保持原来三维模型的形状不变。得到的三维模型中的三角形逼近于正四边形，并且大部分三角形大小近似。

2. 四边形重新网格化

如图 15-6 所示是各种三维模型两种不同尺寸大小的四边形重新网格化结果。从中可以看出，原来的三维模型是三角形面的，重新网格化得到的三维模型是四边形面，但仍能够保持原来三维模型的外观形状不发生很大的变化。得到的三维模型中的四边形逼近于正四边形，并且大部分四边形大小近似。

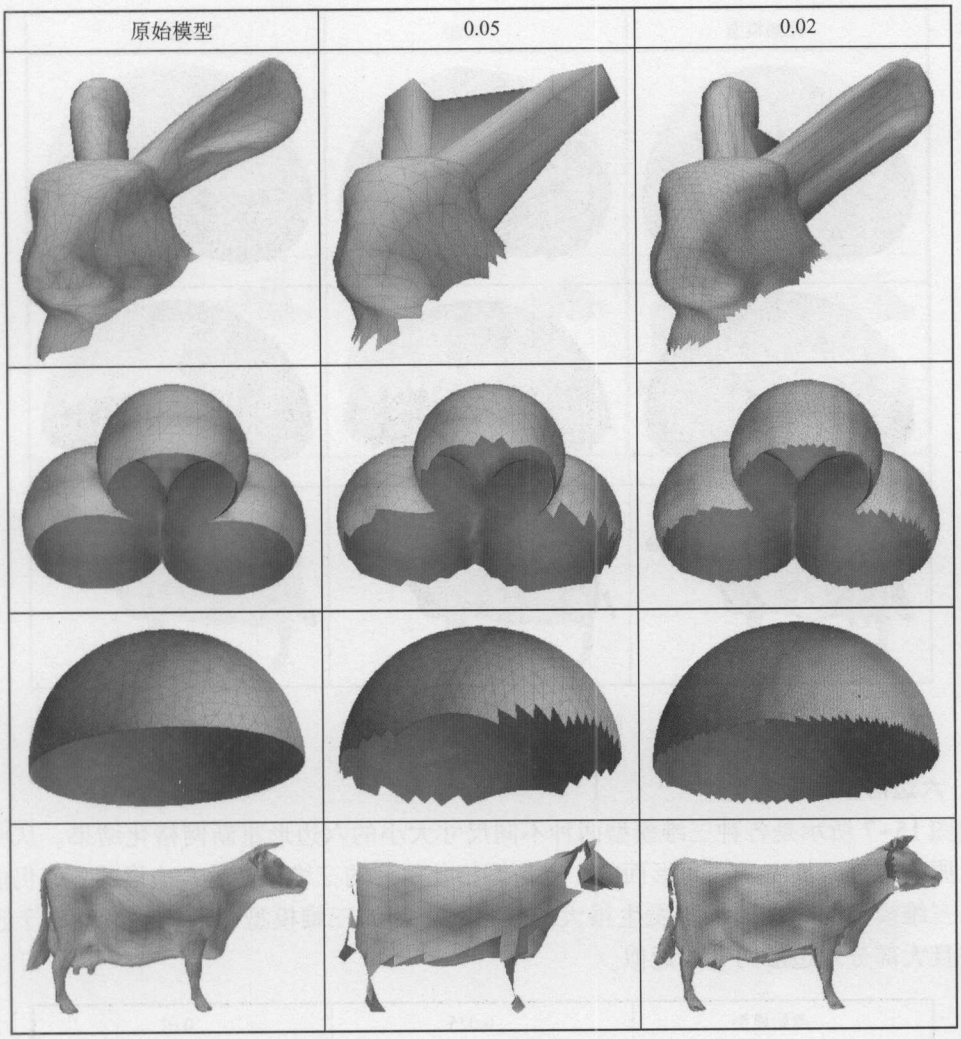


图 15-5　三角形网格

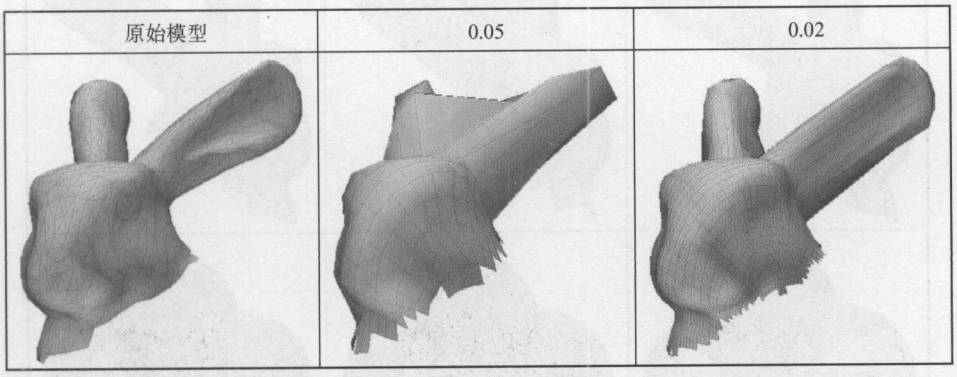


图 15-6　四边形网格

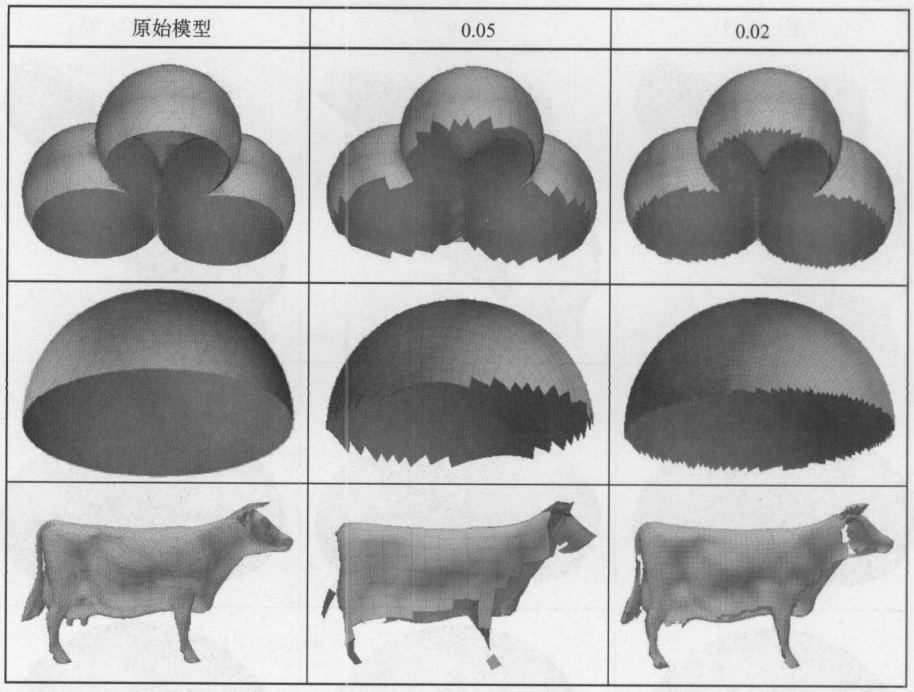


图 15-6　四边形网格（续）

3. 六边形重新网格化

如图 15-7 所示是各种三维模型两种不同尺寸大小的六边形重新网格化结果。从中可以看出，原来的三维模型是三角形面的，重新网格化得到的三维模型是六边形面，但仍能够保持原来三维模型的外观形状不发生很大的变化。得到的三维模型中的六边形逼近于正四边形，并且大部分六边形的大小近似。

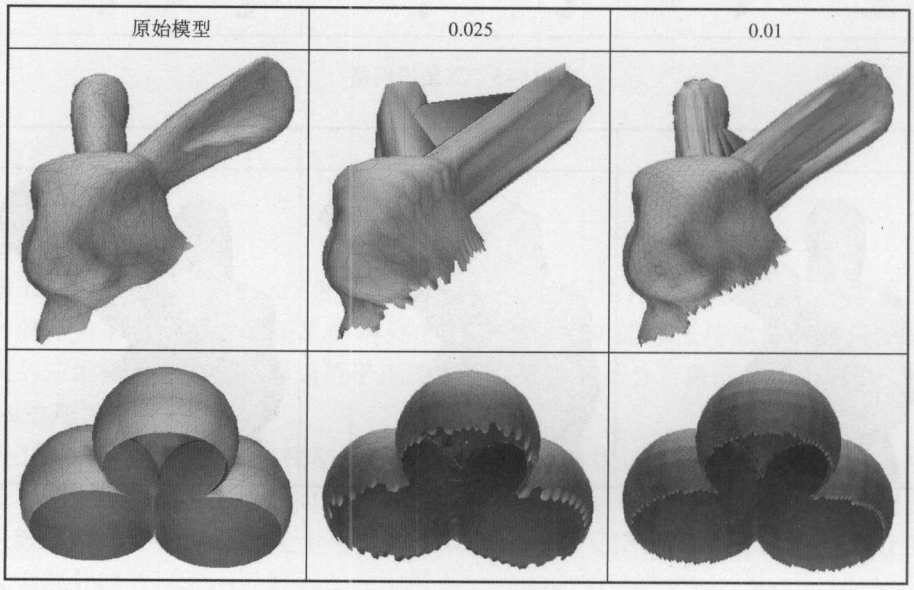


图 15-7　六边形网格

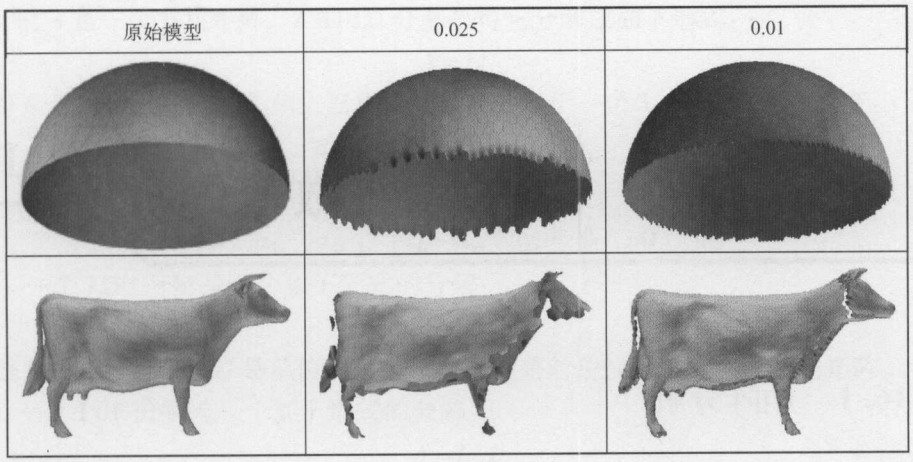


图 15-7　六边形网格（续）

第16章 线性系统数值分析

16.1 矩阵分解

16.1.1 线性系统介绍

三维模型算法的设计涉及大量的数学知识，尤其是线性代数的知识。由于三维模型是一个几何形状，三维模型处理算法还涉及微分几何的知识。在设计算法处理模型时，必须遵守相关的几何定理，也必须依赖相关的几何定理作为指导来进行算法的设计。微分几何定理用到的都是微积分的公式，但由于三维模型是离散的，不是光滑的曲面，因此相应的微积分公式都会被转换为线性系统。从而很多三维模型处理算法最终都可以构建为一个线性系统。线性系统的求解就是三维模型处理算法实现的一个核心要素。线性系统的处理包含两个主要内容：一个是线性系统的求解；一个是矩阵特征向量的求解。线性系统由矩阵和向量构成，因此矩阵的分析是线性系统中一个重要模块。线性系统的求解大多数也是通过矩阵分解得到的。虽然三维模型处理算法实现时都是直接调用相关包装好的线性系统函数库来完成线性系统的求解，但只有了解和掌握这些函数库实现的原理才能设计更高的性能更好的算法。

矩阵分解是将一个矩阵拆解为数个矩阵的乘积。可分为三角分解、满秩分解、QR 分解、Jordan 分解和 SVD（奇异值）分解等。常见的矩阵分解有 3 种：三角分解法（Triangular Factorization）；QR 分解法（QR Factorization）；奇异值分解法（Singular Value Decomposition）。当矩阵的数据量很大时，将一个矩阵分解为若干个矩阵的乘积可以大大降低存储空间。矩阵分解可以减少真正进行问题处理时的计算量，提高算法速度。矩阵分解可以高效和有效地解决某些问题，可以提高算法数值稳定性，这一点对于算法计算来说可以有效地控制误差。

16.1.2 三角分解法

三角分解法也称 LU 分解。在线性代数中，LU 分解是矩阵分解的一种，可以将一个矩阵分解为一个下三角矩阵和一个上三角矩阵的乘积，或者是它们和一个置换矩阵的乘积。LU 分解主要应用在数值分析中，用来解线性方程、求反矩阵或计算行列式。LU 分解法是将原正方（Square）矩阵分解成一个上三角形矩阵或是排列（Permuted）的上三角形矩阵和一个下三角形矩阵，这样的分解法又称为三角分解法。这种分解法所得到的上下三角形矩阵并非唯一，还可找到其他几个不同的一对上下三角形矩阵，此两三角形矩阵相乘也会得到原矩阵。

（1）设 A 是一个方块矩阵。A 的 LU 分解是将它分解成如下形式：

$$A = LU$$

其中 L 和 U 分别是下三角矩阵和上三角矩阵。例如对于一个 3×3 的矩阵，就有：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

（2）一个 LDU 分解是一个如下形式的分解：

$$A = LDU$$

其中 D 是对角矩阵，L 和 U 是单位三角矩阵，也就是对角线上全是 1 的三角矩阵。

（3）一个 LUP 分解是一个如下形式的分解：

$$A = LUP$$

其中 L 和 U 仍是三角矩阵，P 是一个置换矩阵。

（4）一个充分消元的 LU 分解为如下形式：

$$PAQ = LU$$

LU 分解的存在性和唯一性。

一个可逆矩阵可以进行 LU 分解当且仅当它的所有子式都非零。如果要求其中的 L 矩阵或 U 矩阵为单位三角矩阵，那么分解是唯一的。同理可知，矩阵的 LDU 可分解条件也相同，并且总是唯一的。即使矩阵不可逆，LU 仍然可能存在。实际上，如果一个秩为 k 的矩阵的前 k 个顺序主子式不为零，那么它就可以进行 LU 分解，但反之则不然。在任意域上一个方块矩阵可进行 LU 分解的充要条件可以用某些特定子矩阵的秩表示。用高斯消元法来得到 LU 分解的算法也可以扩张到任意域上。

不仅是方块矩阵，任意矩阵 A 都可以进行 LUP 分解。其中的 L 和 P 矩阵是方阵，U 矩阵则与 A 形状一样。对于阶数很大的稀疏矩阵，有简便算法来获得其 LU 分解：这时的 L 和 U 也是稀疏矩阵。从理论上来说，算法的复杂度约等于非零系数的个数，而不是矩阵的大小阶数。这些算法通过运用行和列的交换，使得过程中零系数因为操作而变成非零系数的次数减到最少。

16.1.3　Cholesky 分解

（1）埃尔米特矩阵（Hermitian matrix）或者自共轭矩阵，是共轭对称的方阵。埃尔米特矩阵中每一个第 i 行第 j 列的元素都与第 j 行第 i 列的元素的共轭相等。也就是对于矩阵

$$A = \{a_{i,j}\} \in C^{m \times n} \tag{16-1}$$

中的元素，如果 $\overline{(\cdot)}$ 为共轭算子，满足如下条件：

$$a_{i,j} = \overline{a_{j,i}} \tag{16-2}$$

也可以表示为：

$$A = A^{\mathrm{H}} \tag{16-3}$$

例如，下面的矩阵就是一个埃尔米特矩阵：

$$\begin{bmatrix} 3 & 2+i \\ 2-i & 1 \end{bmatrix}$$

埃尔米特矩阵主对角线上的元素都是实数的，其特征值也是实数。对于只包含实数元素的矩阵，如果它是对称阵，即所有元素关于主对角线对称，那么它也是埃尔米特矩阵。也就是说，实对称矩阵是埃尔米特矩阵的特例。

（2）正定矩阵是埃尔米特矩阵的一种，有时会简称为正定阵。一个 $n \times n$ 的实对称矩阵 M 是正定的当且仅当对于所有的非零实系数向量 z，都有 $z^{\mathrm{T}}Mz>0$。其中 z^{T} 表示 z 的转置。

对于复数的情况，定义则为：一个 $n \times n$ 的埃尔米特矩阵 M 是正定的当且仅当对于每个非零的复向量 z，都有 $z^{*}Mz>0$。其中 z^{*} 表示 z 的共轭转置。由于 M 是埃尔米特矩阵，经计算可知，对于任意的复向量 z，$z^{*}Mz$ 必然是实数，从而可以与 0 比较大小。因此这个定义是自洽的。

① 正定阵的判断条件。

对 $n \times n$ 的埃尔米特矩阵 M，下列性质与“M 为正定矩阵”等价。

- 矩阵 M 的所有的特征值 λ_i 都是正的。
- 存在唯一的下三角矩阵 L，其主对角线上的元素全是正的，使得：

$$M = LL^{*} \tag{16-4}$$

其中 L^{*} 是 L 的共轭转置。这一分解被称为 Cholesky 分解。

② 半正定，负定，半负定。

与正定矩阵相对应的，一个 $n \times n$ 的埃尔米特矩阵 M 是负定矩阵当且仅当对

$$x^{*}Mx<0$$

M 是半正定矩阵当且仅当：

$$x^{*}Mx \geqslant 0$$

M 是半负定矩阵当且仅当：

$$x^{*}Mx \leqslant 0$$

（3）如果矩阵 A 是埃尔米特矩阵，并且是正定矩阵，可以把 A 进行 Cholesky 分解。对每一个正定矩阵，Cholesky 分解都唯一存在。Cholesky 分解是 LU 分解的特例，这里 U 是 L 的转置。但比起一般的 LU 分解，计算 Cholesky 分解更为快捷，并具有更高的数值稳定性。A 的 Cholesky 分解形式为：

$$A = LL^{*} \tag{16-5}$$

这个分解被称作 Cholesky 分解。对每一个正定矩阵，Cholesky 分解都唯一存在。此外，比起一般的 LU 分解，计算 Cholesky 分解更为快捷，并具有更高的数值稳定性。

16.1.4 QR 分解法

QR 分解法是将矩阵分解成一个正规正交矩阵与上三角形矩阵，所以称为 QR 分解法，与此正规正交矩阵的通用符号 Q 有关。实数正交矩阵是方块矩阵 Q，它的转置矩阵是它的逆矩阵：

$$Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = I \tag{16-6}$$

QR 分解法是目前求一般矩阵全部特征值的最有效并广泛应用的方法，一般矩阵先经过正交相似变化成为 Hessenberg 矩阵，然后再应用 QR 方法求特征值和特征向量。它是将矩阵分解成一个正规正交矩阵 Q 与上三角形矩阵 R，所以称为 QR 分解法，与此正规正交矩阵的通用符号 Q 有关。如果 A 是一个 $m \times n$ 的矩阵，当 $m \leqslant n$ 时，其 QR 分解后，Q 为一个 $m \times m$

的矩阵，R 是一个 $m \times n$ 的矩阵。当 $m \geqslant n$ 时，其 QR 分解后，Q 为一个 $m \times n$ 的矩阵，R 是一个 $n \times n$ 的矩阵。QR 分解经常用来解线性最小二乘法问题。QR 分解也是特定特征值算法即 QR 算法的基础。QR 分解的实际计算有很多方法，如 Givens 旋转、Householder 变换，以及 Gram - Schmidt 正交化等。每一种方法都有其优点和缺点。

实数矩阵 A 的 QR 分解是把 A 分解为

$$A = QR \tag{16-7}$$

这里的 Q 是正交矩阵，而 R 是上三角矩阵。类似，我们可以定义 A 的 QL、RQ 和 LQ 分解。我们可以因数分解复数 $m \times n$ 矩阵（有着 $m \geqslant n$）为 $m \times n$ 酉矩阵（在 $Q * Q = I$ 的意义上）和 $n \times n$ 上三角矩阵的乘积。如果 A 是非奇异的，当我们要求 R 的对角是正数时，则这个因数分解结果是唯一的。

16.1.5　奇异值分解法

奇异值分解（Singular Value Decomposition，SVD）是另一种正交矩阵分解法；SVD 是最可靠的分解法，但它比 QR 分解法要花上近 10 倍的计算时间。$[U,S,V] = \mathrm{svd}(A)$，其中 U 和 V 代表二个相互正交矩阵，而 S 代表一对角矩阵。和 QR 分解法相同，原矩阵 A 不必为正方矩阵。使用 SVD 分解法的用途是解最小平方误差法和数据压缩。奇异值分解在某些方面与对称矩阵或 Hermite 矩阵基于特征向量的对角化类似。然而这两种矩阵分解尽管有其相关性，但还是有明显的不同。对称阵特征向量分解的基础是谱分析，而奇异值分解则是谱分析理论在任意矩阵上的推广。

假设 M 是一个 $m \times n$ 阶矩阵，其中的元素全部属于实数域或复数域。如此则存在一个分解使得：

$$M = U \sum V^* \tag{16-8}$$

其中 U 是 $m \times m$ 阶酉矩阵；Σ 是半正定 $m \times n$ 阶对角矩阵；而 V^*，即 V 的共轭转置，是 $n \times n$ 阶酉矩阵。这样的分解就称作 M 的奇异值分解。Σ 对角线上的元素 $\sum i$，i 即为 M 的奇异值。常见的做法是把奇异值由大而小排列。如此 Σ 便能由 M 唯一确定了，但 U 和 V 仍然不能确定。

在矩阵 M 的奇异值分解中

$$M = U \sum V^* \tag{16-9}$$

U 的列组成一套对 M 的正交“输入”或“分析”的基向量。这些向量是 $M * M$ 的特征向量。

V 的列组成一套对 M 的正交“输出”的基向量。这些向量是 $M * M$ 的特征向量。

Σ 对角线上的元素是奇异值，可视为是在输入与输出间进行的标量的“缩放控制”。这些是 $M * M$ 及 $MM *$ 的奇异值，并与 U 和 V 的行向量相对应。

16.2　特征值和特征向量

1. 定义

设 A 是一个 n 阶方阵，λ 是一个数，如果方程

$$AX=\lambda X \tag{16-10}$$

存在非零解向量，则称 λ 为 A 的一个特征值，相应的非零解向量 X 称为属于特征值 λ 的特征向量，上述方程也可写成

$$(A-\lambda E)=0 \tag{16-11}$$

这是 n 个未知数 n 个方程的齐次线性方程组，它有非零解的充分必要条件是系数行列式

$$|A-\lambda E|=0 \tag{16-12}$$

即

$$\begin{vmatrix} a_{11}-\lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-\lambda & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn}-\lambda \end{vmatrix}=0 \tag{16-13}$$

上式是以 λ 为未知数的一元 n 次方程，称为方阵 A 的特征方程. 其左端 $|A-\lambda E|$ 是 λ 的 n 次多项式，记作 $f(\lambda)$，称为方阵 A 的特征多项式。

$$\begin{aligned} f(\lambda)=|A-\lambda E| &= \begin{vmatrix} a_{11}-\lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-\lambda & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn}-\lambda \end{vmatrix} \\ &=(-1)^n\lambda^n+a_1\lambda^{n-1}+\cdots+a_{n-1}\lambda+a_n \end{aligned} \tag{16-14}$$

显然，A 的特征值就是特征方程的解。特征方程在复数范围内恒有解，其个数为方程的次数（重根按重数计算），因此，n 阶矩阵 A 有 n 个特征值。

设 n 阶矩阵 $A=(a_{ij})$ 的特征值为 $\lambda_1,\lambda_2,\cdots,\lambda_n$，由多项式的根与系数之间的关系，不难证明

$$\lambda_1+\lambda_2+\cdots+\lambda_n=a_{11}+a_{22}+\cdots+a_{nn} \tag{16-15}$$

$$\lambda_1\lambda_2\cdots\lambda_n=|A| \tag{16-16}$$

若 λ 为 A 的一个特征值，则 λ 一定是方程 $|A-\lambda E|=0$ 的根，因此又称特征根，若 λ 为方程 $|A-\lambda E|=0$ 的 n_i 重根，则 λ 称为 A 的 n_i 重特征根。方程 $(A-\lambda E)X=0$ 的每一个非零解向量都是相应于 λ 的特征向量，于是我们可以得到求矩阵 A 的全部特征值和特征向量的方法。

第一步：计算 A 的特征多项式 $|A-\lambda E|$。

第二步：求出特征方程 $|A-\lambda E|=0$ 的全部根，即为 A 的全部特征值。

第三步：对于 A 的每一个特征值 λ，求出齐次线性方程组。

$$(A-\lambda E)X=0 \tag{16-17}$$

的一个基础解系 $\xi_1,\xi_2,\cdots,\xi_s$，则 A 的属于特征值 λ 的全部特征向量是 $k_1\xi_1+k_2\xi_2+\cdots+k_s\xi_s$（其中 $k_1,k_2,\cdots,k_s$ 是不全为零的任意实数）。

2. 实例

例一：求 $A=\begin{pmatrix} 3 & -1 \\ -1 & 3 \end{pmatrix}$ 的特征值和特征向量。

解 A 的特征多项式为

$$|A-\lambda E| = \begin{vmatrix} 3-\lambda & -1 \\ -1 & 3-\lambda \end{vmatrix} = (3-\lambda)^2 - 1 = (4-\lambda)(2-\lambda)$$

所以 A 的特征值为 $\lambda_1 = 2, \lambda_2 = 4$。

当 $\lambda_1 = 2$ 时，解齐次线性方程组 $(A-2E)X = 0$ 得

$$\begin{cases} x_1 - x_2 = 0 \\ -x_1 + x_2 = 0 \end{cases}$$

解得 $x_1 = x_2$，令 $x_2 = 1$，则其基础解系为：$\xi_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$。

因此，属于 $\lambda_1 = 2$ 的全部特征向量为：$k_1\xi_1 (k_1 \neq 0)$。

当 $\lambda_2 = 4$ 时，解齐次线性方程组 $(A-4E)X = 0$ 得 $x_1 = -x_2$，令 $x_2 = 1$，则其基础解系为：$\xi_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$，因此 A 的属于 $\lambda_2 = 4$ 的全部特征向量为 $k_2\xi_2 (k_2 \neq 0)$。

若 ξ 是 A 的属于 λ 的特征向量，则 $k\xi (k \neq 0)$ 也是对应于 λ 的特征向量，因此特征向量不能由特征值唯一确定。反之，不同特征值对应的特征向量不会相等，即一个特征向量只能属于一个特征值。

例二： 求矩阵 $A = \begin{pmatrix} 1 & -2 & 2 \\ -2 & -2 & 4 \\ 2 & 4 & -2 \end{pmatrix}$ 的特征值和特征向量。

解　A 的特征多项式为

$$|A-\lambda E| = \begin{vmatrix} 1-\lambda & -2 & 2 \\ -2 & -2-\lambda & 4 \\ 2 & 4 & -2-\lambda \end{vmatrix} = -(\lambda-2)^2(\lambda+7)$$

所以 A 的特征值为 $\lambda_1 = \lambda_2 = 2$（二重根），$\lambda_3 = -7$。

对于 $\lambda_1 = \lambda_2 = 2$，解齐次线性方程组 $(A-2E)X = 0$。由

$$A-2E = \begin{pmatrix} -1 & -2 & 2 \\ -2 & -4 & 4 \\ 2 & 4 & -4 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & -2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

得基础解系为：$\xi_1 = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}, \xi_2 = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$

因此，属于 $\lambda_1 = \lambda_2 = 2$ 的全部特征向量为：$k_1\xi_1 + k_2\xi_2$（k_1, k_2 不同时为零）。

对于 $\lambda_3 = -7$，解齐次线性方程组 $(A+7E)X = 0$。由

$$A+7E = \begin{pmatrix} 8 & -2 & 2 \\ -2 & 5 & 4 \\ 2 & 4 & 5 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & \frac{1}{2} \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

得基础解系为：$\xi_3 = \begin{pmatrix} 1 \\ 2 \\ -2 \end{pmatrix}$。

因此，属于 $\lambda_3=-7$ 的全部特征向量为：$k_3\xi_3(k_3\neq 0)$。

对于方阵 A 的每一个特征值，都可以求出其全部的特征向量，但对于属于不同特征值的特征向量，它们之间的关系如下。

定理 属于不同特征值的特征向量一定线性无关。

证明 设 $\lambda_1,\lambda_2,\cdots,\lambda_m$ 是矩阵 A 的不同特征值，而 $\xi_1,\xi_2,\cdots,\xi_m$ 分别是属于 $\lambda_1,\lambda_2,\cdots,\lambda_m$ 的特征向量，要证 $\xi_1,\xi_2,\cdots,\xi_m$ 是线性无关的。我们对特征值的个数 m 作数学归纳法证明。

当 $m=1$ 时，由于特征向量不为零，所以结论显然成立。

当 $m>1$ 时，假设 $m-1$ 时结论成立。

由于 $\lambda_1,\lambda_2,\cdots,\lambda_m$ 是 A 的不同特征值，而 ξ_i 是属于 λ_i 的特征向量，因此

$$A\xi_i=\lambda_i\xi_i(i=1,2,\cdots,m)$$

如果存在一组实数 $k_1,k_2,\cdots,k_m$ 使

$$k_1\xi_1+k_2\xi_2+\cdots+k_m\xi_m=0$$

则上式两边乘以 λ_m 得

$$k_1\lambda_m\xi_1+k_2\lambda_m\xi_2+\cdots+k_m\lambda_m\xi_m=0$$

另一方面，$A(k_1\xi_1+k_2\xi_2+\cdots+k_m\xi_m)=0$，即

$$k_1\lambda_1\xi_1+k_2\lambda_2\xi_2+\cdots+k_m\lambda_m\xi_m=0$$

那么：

$$k_1(\lambda_m-\lambda_1)\xi_1+k_2(\lambda_m-\lambda_2)\xi_2+\cdots+k_{m-1}(\lambda_m-\lambda_{m-1})\xi_{m-1}=0$$

由归纳假设，$\xi_1,\xi_2,\cdots,\xi_{m-1}$ 线性无关，因此

$$k_i(\lambda_m-\lambda_i)=0(i=1,2,\cdots,m-1)$$

而 $\lambda_1,\lambda_2,\cdots,\lambda_m$ 互不相同，所以 $k_i=0(i=1,2,\cdots,m-1)$，于是 $k_m\xi_m=0$。

因 $\xi_m\neq 0$，于是 $k_m=0$。可见 $\xi_1,\xi_2,\cdots,\xi_m$ 线性无关。

16.3 相似矩阵

1. 定义

设 A、B 都是 n 阶方阵，若存在满秩矩阵 P，使得

$$B=P^{-1}AP \tag{16-18}$$

则称 A 与 B 相似，记作 $A\sim B$，且满秩矩阵 P 称为将 A 变为 B 的相似变换矩阵。

“相似”是矩阵间的一种关系，这种关系具有如下性质。

(1) 反身性：$A\sim A$。

(2) 对称性：若 $A\sim B$，则 $B\sim A$。

(3) 传递性：若 $A\sim B$，$B\sim C$，则 $A\sim C$。

定理一 相似矩阵有相同的特征多项式，因此有相同的特征值。

证明 设 $A\sim B$，则存在满秩矩阵 P，使 $B=P^{-1}AP$

于是

$$\begin{aligned}|B-\lambda E|&=|P^{-1}AP-\lambda E|=|P^{-1}AP-P^{-1}(\lambda E)P|\\&=|P^{-1}(A-\lambda E)P|=|A-\lambda E|\end{aligned}$$

推论　若 n 阶矩阵 $\boldsymbol{A}$ 与对角矩阵

$$\Lambda=\begin{pmatrix}\lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n\end{pmatrix}$$

相似，则 $\lambda_1,\lambda_2,\cdots,\lambda_n$ 即是 A 的 n 个特征值。

定理二　设 ξ 是矩阵 A 的属于特征值 λ_0 的特征向量，且 $A\sim B$，即存在满秩矩阵 P 使 $B=P^{-1}AP$，则 $\eta=P^{-1}\xi$ 是矩阵 B 的属于 λ_0 的特征向量。

证明　因 ξ 是矩阵 A 的属于特征值 λ_0 的特征向量，则有 $A\xi=\lambda_0\xi$

于是

$$\begin{aligned}B\eta&=(P^{-1}AP)(P^{-1}\xi)=P^{-1}A\xi\\&=P^{-1}\lambda_0\xi=\lambda_0p^{-1}\xi=\lambda_0\eta\end{aligned}$$

所以 η 是矩阵 B 的属于 λ_0 的特征向量。

下面我们要讨论的主要问题是：对 n 阶矩阵 A，寻求相似变换矩阵 P，使 $P^{-1}AP=\Lambda$ 为对角矩阵，这就称为把方阵 A 对角化。

定理三　n 阶矩阵 A 与对角矩阵 $\Lambda=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)$ 相似的充分必要条件是：矩阵 A 有 n 个线性无关的分别属于特征值 $\lambda_1,\lambda_2,\cdots,\lambda_n$ 的特征向量，$\lambda_1,\lambda_2,\cdots,\lambda_n$ 中可以有相同的值。

证明

必要性

设 A 与对角矩阵 $\Lambda=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)$ 相似，则存在满秩矩阵 P，使

$$P^{-1}AP=\Lambda=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)$$

设 $P=(\xi_1,\xi_2,\cdots,\xi_n)$，则由上式得

$$AP=P\Lambda$$

即 $A(\xi_1,\xi_2,\cdots,\xi_n)=(\xi_1,\xi_2,\cdots,\xi_n)\Lambda=(\lambda_1\xi_1,\lambda_2\xi_2,\cdots,\lambda_n\xi_n)$。

因此 $A\xi_i=\lambda_i\xi_i$

λ_i 是 A 的特征值，ξ_i 是 A 的属于 λ_i 的特征向量，又因 P 是满秩的，故 $\xi_1,\xi_2,\cdots,\xi_n$ 线性无关。

充分性

如果 A 有 n 个线性无关的分别属于特征值 $\lambda_1,\lambda_2,\cdots,\lambda_n$ 的特征向量 $\xi_1,\xi_2,\cdots,\xi_n$，

则有

$$A\xi_i=\lambda_i\xi_i(i=1,2,\cdots,n)$$

设 $P=(\xi_1,\xi_2,\cdots,\xi_n)$ 则 P 是满秩的，于是

$$\begin{aligned}AP&=A(\xi_1,\xi_2,\cdots,\xi_n)=(A\xi_1,A\xi_2,\cdots,A\xi_n)=(\lambda_1\xi_1,\lambda_2\xi_2,\cdots,\lambda_n\xi_n)\\&=P\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)\end{aligned}$$

即 $P^{-1}AP=\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)$。

由以上定理可以知道，一个 n 阶方阵能否与一个 n 阶对角矩阵相似，关键在于它是否有 n 个线性无关的特征向量。

(1) 如果一个 n 阶方阵有 n 个不同的特征值，则由定理可知，它一定有 n 个线性无关的

特征向量，因此该矩阵一定相似于一个对角矩阵。

（2）如果一个 n 阶方阵有 n 个特征值，其中有重复的，则我们可分别求出属于每个特征值的基础解系，如果每个 n_i 重特征值的基础解系含有 n_i 个线性无关的特征向量，则该矩阵与一个对角矩阵相似，否则该矩阵不与一个对角矩阵相似。

可见，如果一个 n 阶方阵有 n 个线性无关的特征向量，则该矩阵与一个 n 阶对角矩阵相似，并且以这 n 个线性无关的特征向量作为列向量构成的满秩矩阵 P，使 $P^{-1}AP=\Lambda$ 为对角矩阵，而对角线上的元素就是这些特征向量顺序对应的特征值。

2. 实例

例一 设矩阵 $A=\begin{Bmatrix}4&0&0\\0&3&1\\0&1&3\end{Bmatrix}$，求一个满秩矩阵 P，使 $P^{-1}AP$ 为对角矩阵。

解 A 的特征多项式为

$$|A-\lambda E|=\begin{vmatrix}4-\lambda&0&0\\0&3-\lambda&1\\0&1&3-\lambda\end{vmatrix}=(\lambda-4)^2(2-\lambda)$$

所以 A 的特征值为 $\lambda_1=\lambda_2=4,\lambda_3=2$。

对于 $\lambda_1=\lambda_2=4$ 解齐次线性方程组 $(A-4E)X=0$，得基础解系 $\xi_1=(1,0,0)^{\mathrm{T}},\xi_2=(0,1,1)^{\mathrm{T}}$，即为 A 的两个特征向量 ξ_1,ξ_2。

对于 $\lambda_3=2$，解齐次线性方程组 $(A-2E)X=0$，得基础解系 $\xi_3=(0,-1,1)^{\mathrm{T}}$，即为 A 的一个特征向量 ξ_3。

显然 ξ_1,ξ_2,ξ_3 是线性无关的，取

$$P=\begin{pmatrix}1&0&0\\0&1&-1\\0&1&1\end{pmatrix}$$

即有

$$P^{-1}AP=\begin{pmatrix}4&0&0\\0&4&0\\0&0&2\end{pmatrix}$$

例二 设 $A=\begin{pmatrix}3&1&0\\-4&-1&0\\4&-8&-2\end{pmatrix}$，考虑 A 是否相似于对角矩阵。

解

$$|A-\lambda E|=\begin{vmatrix}3-\lambda&1&0\\-4&-1-\lambda&0\\4&-8&-2-\lambda\end{vmatrix}=-(\lambda-1)^2(\lambda+2)$$

所以 A 的特征值为 $\lambda_1=\lambda_2=1,\lambda_3=-2$。

对于 $\lambda_1=\lambda_2=1$ 解齐次线性方程组 $(A-E)X=0$，得基础解系 $\xi_1=(3,-6,20)^{\mathrm{T}}$ 即为 A 一个特征向量 ξ_1。

对于 $\lambda_3=-2$，解齐次线性方程组 $(A+2E)X=0$，得基础解系 $\xi_2=(0,0,1)^{\mathrm{T}}$ 即为 A 的另一个特征向量 ξ_2。

由于 A 只有两个线性无关的特征向量，因此 A 不能相似于一个对角矩阵。

16.4　向量组的正交性

在解析几何中，二维、三维向量的长度及夹角等度量性质都可以用向量的内积来表示，现在我们把内积推广到 n 维向量中。

定义　设有 n 维向量 $\alpha=(a_1,a_2,\cdots,a_n)$，$\beta=(b_1,b_2,\cdots b_n)$，令 $(\alpha,\beta)=a_1b_1+a_2b_2+\cdots+a_nb_n$，则 (α,β) 称为向量 α 和 β 的内积。

内积是向量的一种运算，若用矩阵形式表示，当 α 和 β 是行向量时，$(\alpha,\beta)=\alpha\beta^{\mathrm{T}}=\beta\alpha^{\mathrm{T}}$，当 α 和 β 都是列向量时，$(\alpha,\beta)=\alpha^{\mathrm{T}}\beta=\beta^{\mathrm{T}}\alpha$。

内积具有下列性质（其中 α、β、γ 为 n 维向量，λ 为常数）。

（1）$(\alpha,\beta)=(\beta,\alpha)$。

（2）$(\lambda\alpha,\beta)=\lambda(\alpha,\beta)$。

（3）$(\alpha+\beta,\gamma)=(\alpha,\gamma)+(\beta,\gamma)$。

（4）$(\alpha,\alpha)\geqslant 0$，当且仅当 $\alpha=0$ 时等号成立。

定义　令 $|\alpha|=\sqrt{(\alpha,\alpha)}=\sqrt{a_1^2+a_2^2+\cdots+a_n^2}$，称 $|\alpha|$ 为 n 维向量 α 的模（或长度）。

向量的模具有如下性质。

（1）当 $\alpha\neq 0$ 时，$|\alpha|>0$；当 $\alpha=0$ 时，$|\alpha|=0$。

（2）$|\lambda\alpha|=|\lambda|\,|\alpha|$，（$\lambda$ 为实数）。

（3）$|(\alpha,\beta)|\leqslant|\alpha|\,|\beta|$。

（4）$|\alpha+\beta|\leqslant|\alpha|+|\beta|$。

特别地，当 $|\alpha|=1$ 时，称 α 为单位向量。

如果 $|\alpha|\neq 0$，由向量的模具性质（2），向量 $\dfrac{1}{|\alpha|}\alpha$ 是一个单位向量。可见，用向量 α 的模去除向量 α，可得到一个与 α 同向的单位向量，我们称这一运算为向量的单位化或标准化。

如果 α、β 都为非零向量，由向量的模具性质（3）$\dfrac{|(\alpha,\beta)|}{|\alpha|\cdot|\beta|}\leqslant 1$，于是有下述定义。

（1）当 $|\alpha|\neq 0$、$|\beta|\neq 0$ 时，$\theta=\arccos\dfrac{(\alpha,\beta)}{|\alpha|\cdot|\beta|}$ 称为 n 维向量 α、β 的夹角。

特别地：当 $(\alpha,\beta)=0$ 时，$\theta=\dfrac{\pi}{2}$。

（2）当 $(\alpha,\beta)=0$ 时，称向量 α 与 β 正交（显然，若 $\alpha=0$，则 α 与任何向量都正交）。向量的正交性可推广到多个向量的情形。

（3）已知 m 个非零向量 $\alpha_1,\alpha_2,\cdots,\alpha_m$，若 $(\alpha_i,\alpha_j)=0(i,j=1,2,\cdots,m,i\neq j)$，则称 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 为正交向量组。

（4）若向量组 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 为正交向量组，且 $|\alpha_i|=1(i=1,2,\cdots,m)$，则称 $\alpha_1,$

$\alpha_2,\cdots,\alpha_m$ 为标准正交向量组。

例如，n 维单位向量组 $\varepsilon_1=(1,0,\cdots,0),\varepsilon_2=(0,1,\cdots,0),\cdots,\varepsilon_n=(0,0,\cdots,1)$ 是正交向量组。正交向量组有下述重要性质。

定理一

（1）正交向量组 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 是线性无关的向量组。

定理的逆命题一般不成立，但任意线性无关的向量组总可以通过如下所述的正交化过程，构成正交化向量组，进而通过单位化，构成标准正交向量组。

（2）设向量组 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 线性无关，由此可做出含有 m 个向量的正交向量组 $\beta_1,\beta_2,\cdots,\beta_m$，其中：

$$\beta_1=\alpha_1$$

$$\beta_2=\alpha_2-\frac{(\alpha_2,\beta_1)}{(\beta_1,\beta_1)}\beta_1$$

$$\cdots\cdots$$

$$\beta_m=\alpha_m-\frac{(\alpha_m,\beta_1)}{(\beta_1,\beta_1)}\beta_1-\frac{(\alpha_m,\beta_2)}{(\beta_2,\beta_2)}\beta_2-\cdots-\frac{(\alpha_m,\beta_{m-1})}{(\beta_{m-1},\beta_{m-1})}\beta_{m-1} \tag{16-19}$$

再取 $\eta_i=\dfrac{\beta_i}{|\beta_i|}(i=1,2,\cdots,m)$，则 $\eta_1,\eta_2,\cdots,\eta_m$ 为标准正交向量组。

上述从线性无关向量组 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 导出正交向量组 $\beta_1,\beta_2,\cdots,\beta_m$ 的过程称为施密特（Schimidt）正交化过程。它不仅满足 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 与 $\alpha_1,\alpha_2,\cdots,\alpha_m$ 等价，还满足对任何 $k(1\leqslant k\leqslant m)$，向量组 $\beta_1,\beta_2,\cdots,\beta_k$ 与 $\alpha_1,\alpha_2,\cdots,\alpha_k$ 等价。

例一 把向量组 $\alpha_1=(1,1,0,0),\alpha_2=(1,0,1,0),\alpha_3=(-1,0,0,1)$ 化为标准正交向量组。

解 容易验证 $\alpha_1,\alpha_2,\alpha_3$ 是线性无关的。

将 $\alpha_1,\alpha_2,\alpha_3$ 正交化，令

$$\beta_1=\alpha_1=(1,1,0,0)$$

$$\beta_2=\alpha_2-\frac{(\alpha_2,\beta_1)}{(\beta_1,\beta_1)}\beta_1=(1,0,1,0)-\frac{1}{2}(1,1,0,0)=\left(\frac{1}{2},-\frac{1}{2},1,0\right)$$

$$\beta_3=\alpha_3-\frac{(\alpha_3,\beta_1)}{(\beta_1,\beta_1)}\beta_1-\frac{(\alpha_3,\beta_2)}{(\beta_2,\beta_2)}\beta_2$$

$$=(-1,0,0,1)-\left(-\frac{1}{2}\right)(1,1,0,0)-\left(-\frac{1}{3}\right)\left(\frac{1}{2},-\frac{1}{2},1,0\right)$$

$$=\left(-\frac{1}{3},\frac{1}{3},\frac{1}{3},1\right)$$

再把 β_1,β_2,β_3 单位化

$$\eta_1=\frac{\beta_1}{|\beta_1|}=\frac{1}{\sqrt{2}}(1,1,0,0)=\left(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}},0,0\right)$$

$$\eta_2=\frac{\beta_2}{|\beta_2|}=\frac{1}{\sqrt{\frac{3}{2}}}\left(\frac{1}{2},-\frac{1}{2},1,0\right)=\left(\frac{1}{\sqrt{6}},-\frac{1}{\sqrt{6}},\frac{2}{\sqrt{6}},0\right)$$

$$\eta_3=\frac{\beta_3}{|\beta_3|}=\frac{1}{\sqrt{\frac{4}{3}}}\left(-\frac{1}{3},\frac{1}{3},\frac{1}{3},1\right)=\left(-\frac{1}{2\sqrt{3}},\frac{1}{2\sqrt{3}},\frac{1}{2\sqrt{3}},\frac{\sqrt{3}}{2}\right)$$

则 η_1,η_2,η_3 即为所求的标准正交向量组。

定理二　若 $\alpha_1,\alpha_2,\cdots,\alpha_r$ 是 n 维正交向量组，$r<n$，则必有 n 维非零向量 X，使 $\alpha_1,\alpha_2,\cdots,\alpha_r$，$X$ 成为正交向量组。

推论　含有 r 个$(r<n)$向量的 n 维正交（或标准正交）向量组，总可以添加 $n-r$ 个 n 维非零向量，构成含有 n 个向量的 n 维正交向量组。

例二　已知 $\alpha_1=\begin{pmatrix}1\\1\\1\end{pmatrix}$，求一组非零向量 α_2,α_3，使 $\alpha_1,\alpha_2,\alpha_3$ 成为正交向量组。

解　α_2,α_3 应满足方程 $\alpha_1^{\mathrm{T}}x=0$，即

$$x_1+x_2+x_3=0$$

它的基础解系为

$$\xi_1=\begin{pmatrix}1\\0\\-1\end{pmatrix},\xi_2=\begin{pmatrix}0\\1\\-1\end{pmatrix}$$

把基础解系正交化，即为所求。亦即取

$$\alpha_2=\xi_1,\alpha_3=\xi_2-\frac{(\xi_2,\xi_1)}{(\xi_1,\xi_1)}\xi_1$$

其中$(\xi_2,\xi_1)=1,(\xi_1,\xi_1)=2$，于是得

$$\alpha_2=\begin{pmatrix}1\\0\\-1\end{pmatrix},\alpha_3=\begin{pmatrix}0\\1\\-1\end{pmatrix}-\frac{1}{2}\begin{pmatrix}1\\0\\-1\end{pmatrix}=\begin{pmatrix}-\frac{1}{2}\\1\\-\frac{1}{2}\end{pmatrix}$$

1. 正交矩阵

定义　如果 n 阶矩阵 A 满足 $A^{\mathrm{T}}A=E$（即 $A^{-1}=A^{\mathrm{T}}$），那么称 A 为正交矩阵。

正交矩阵具有如下性质。

(1) 矩阵 A 为正交矩阵的充分必要条件是 $A^{-1}=A^{\mathrm{T}}$。

(2) 正交矩阵的逆矩阵是正交矩阵。

(3) 两个正交矩阵的乘积仍是正交矩阵。

(4) 正交矩阵是满秩的，且 $|=1|A|=1$ 或 -1。

由等式 $A^{\mathrm{T}}A=E$ 可知，正交矩阵 $A=(a_{ij})$ 的元素满足关系式

$$\sum_{k=1}^{n}a_{ik},a_{jk}=\begin{cases}1,i=j\\0,i\neq j\end{cases}\sum_{k=1}^{n}a_{ki},a_{kj}=\begin{cases}1,i=j\\0,i\neq j\end{cases}(\text{其中 } i,j=1,2,\cdots,n)$$

可见正交矩阵任意不同两行（列）对应元素乘积之和为 0，同一行（列）元素的平方和为 1，因此正交矩阵的行（列）所构成的向量组为标准正交向量组，反之亦然。一个 n 阶矩阵为正交矩阵的充分必要条件是它的行（或列）向量组是一个标准正交向量组。

2. 正定矩阵

对二次型 $x^{\mathrm{T}}Ax$，如对任何 $x\neq 0$，恒有 $x^{\mathrm{T}}Ax>0$，则称二次型 $x^{\mathrm{T}}Ax$ 是**正定二次型**。正定二次型的矩阵 A 称为**正定矩阵**。

例如，二元二次型 $f(x_1,x_2)=x_1^2+3x_2^2$，对任何 $x=\begin{bmatrix}t\\u\end{bmatrix}\neq 0$，总有 $f(t,u)=t^2+3u^2>0$，所以 f 是正定二次型，$A=\begin{bmatrix}1&\\&3\end{bmatrix}$ 是正定矩阵。而三元二次型 $f(x_1,x_2,x_3)=x_1^2+3x_2^2$，对 $x=\begin{bmatrix}0\\0\\1\end{bmatrix}\neq 0$，有 $f(0,0,1)=0$，此二次型不是正定二次型，也即 $A=\begin{bmatrix}1&&\\&3&\\&&0\end{bmatrix}$ 不是正定矩阵。

3. 二次型正定的充分必要条件

n 元二次型正定

$\Leftrightarrow x^{\mathrm{T}}Ax$ 的正惯性指数 $p=n$。

$\Leftrightarrow A$ 与 E 合同，即有可逆矩阵 C，使 $C^{\mathrm{T}}AC=E$。

$\Leftrightarrow A$ 的所有特征值全大于零。

$\Leftrightarrow A$ 的顺序主子式全大于零。

$\Leftrightarrow$ 存在可逆矩阵 C，使得 $A=C^{\mathrm{T}}C$。

正定的必要条件如下。

$$a_{ii}>0(i=1,2,\cdots,n);$$
$$|A|>0$$

4. 实对称矩阵的相似对角化

定理三

（1）实对称矩阵的特征值恒为实数，从而它的特征向量都可取为实向量。

（2）实对称矩阵的不同特征值的特征向量是正交的。

（3）正定矩阵的特征值都大于零，为正数。

证明 设 λ_1,λ_2 是实对称矩阵 A 的两个不同的特征值，即 $\lambda_1\neq\lambda_2$。ξ_1,ξ_2 是分别属于 λ_1,λ_2 的特征向量，则

$$A\xi_1=\lambda_1\xi_1,\quad A\xi_2=\lambda_2\xi_2 \tag{16-20}$$

根据内积的性质有

$$(A\xi_1,\xi_2)=(\lambda_1\xi_1,\xi_2)=\lambda_1(\xi_1,\xi_2),$$
$$(A\xi_1,\xi_2)=(A\xi_1)^T\xi_2=\xi_1^TA^T\xi_2=\xi_1^TA\xi_2$$
$$=\xi_1^T\lambda_2\xi_2=\lambda_2(\xi_1,\xi_2)$$

所以 $(\lambda_1-\lambda_2)(\xi_1,\xi_2)=0$，因 $\lambda_1\neq\lambda_2$，故 $(\xi_1,\xi_2)=0$，即 ξ_1 与 ξ_2 正交。

定理四 设 A 为 n 阶对称矩阵，λ 是 A 的特征方程的 r 重根，则矩阵 $A-\lambda E$ 的秩 $R(A-\lambda E)=n-r$，从而对应特征值 λ 恰有 r 个线性无关的特征向量。

定理五 设 A 为 n 阶对称矩阵，则必有正交矩阵 P，使 $P^{-1}AP=\Lambda$，其中 Λ 是以 A 的 n 个特征值为对角元素的对角矩阵。

实例

例三　设 $A=\begin{pmatrix}4&0&0\\0&3&1\\0&1&3\end{pmatrix}$ 求一个正交矩阵 P，使 $P^{-1}AP=\Lambda$ 为对角矩阵。

解　$|A-\lambda E|=\begin{vmatrix}4-\lambda&0&0\\0&3-\lambda&1\\0&1&3-\lambda\end{vmatrix}=(2-\lambda)(4-\lambda)^2$

所以 A 的特征值 $\lambda_1=2,\lambda_2=\lambda_3=4$。

对于 $\lambda_1=2$，解齐次线性方程组 $(A-2E)X=0$，得基础解系

$$\xi_1=\begin{pmatrix}0\\1\\-1\end{pmatrix}$$

因此属于 $\lambda_1=2$ 的标准特征向量为

$$\eta_1=\begin{pmatrix}0\\\dfrac{1}{\sqrt{2}}\\-\dfrac{1}{\sqrt{2}}\end{pmatrix}$$

对于 $\lambda_2=\lambda_3=4$，解齐次线性方程组 $(A-4E)X=0$，得基础解系

$$\xi_2=\begin{pmatrix}1\\0\\0\end{pmatrix},\ \xi_3=\begin{pmatrix}0\\1\\1\end{pmatrix}$$

这两个向量恰好正交，将其单位化即得两个属于 $\lambda=4$ 的标准正交向量

$$\eta_2=\begin{pmatrix}1\\0\\0\end{pmatrix},\ \eta_3=\begin{pmatrix}0\\\dfrac{1}{\sqrt{2}}\\\dfrac{1}{\sqrt{2}}\end{pmatrix}$$

于是得正交矩阵

$$P=(\eta_1,\eta_2,\eta_3)=\begin{pmatrix}0&1&0\\\dfrac{1}{\sqrt{2}}&0&\dfrac{1}{\sqrt{2}}\\-\dfrac{1}{\sqrt{2}}&0&\dfrac{1}{\sqrt{2}}\end{pmatrix}$$

易验证

$$P^{-1}AP=\begin{pmatrix}2&&\\&4&\\&&4\end{pmatrix}$$

第17章 线性系统函数库

17.1 函数库介绍

17.1.1 函数库种类

在实现参数化算法过程中，需要进行求解线性系统和特征值等计算。这些计算依赖于复杂的数值分析计算算法。求解线性系统和特征向量虽然理论上具有各种简单、直观的方法，但在计算机上并不适用。由于计算机是二进制存储数据，因此数据有误差。理论上在求解线性系统和特征向量时，有各种各样的方法，但计算机计算时需要选择误差比较小，速度比较快的一种。如何控制误差，快速、准确地得到结果，这些都是专门的数值分析需要研究的问题。因此需要采用特殊设计的函数库。下面介绍一些通用的，性能比较高的用于线性计算和特征向量计算的函数库。这些函数库都是用 C、C ++ 编程语言实现的，因此需要把它们用 C#语言封装起来。前几章的各种参数化算法的实现线性系统求解和特征向量计算部分就调用了用 C#封装的相应函数库。

（1）SuiteSparseQR 是一个提供多线程，并行计算的大型矩阵 QR 分解方法的函数库。采用 C ++ 语言编写，提供 MATLAB、C、C ++ 接口。支持实数和复数矩阵。SuiteSparseQR 依赖的其他函数库：AMD、COLAMD、CHOLMOD、METIS、CCAMD 和 CCOLAMD。

（2）CHOLMOD 是一个进行 Cholesky 分解的 C 语言函数库，提供 MATLAB 接口。由佛罗里达大学 T. A. Daivs 开发，第一版发布于 2004 年，当前版本 2. 0. 1 支持 CUDA 框架下的 GPU 计算。

（3）UMFPACK（Unsymmetric Multifrontal sparse lu Factorization PACKage）是一个解非对称稀疏线性方程的函数库，C 语言开发，具有 MATLAB 接口。由佛罗里达大学 T. A. Daivs 开发，UMFPACK 最早发布于 1994 年，当前版本为 5. 6. 1。

（4）SuperLU（Supernodal LU）是用来通过 LU 分解对大型、稀疏、非奇异线性系统进行直接求解的函数库。用 C 语言编写，提供 C 和 Fortran 语言接口。线性系统中的矩阵不需要是对称或正定的。SuperLU 主要处理方阵，但也可以对非方阵进行处理。第一版发布于 1997 年，当前版本 4. 3 支持分布计算。

（5）TAUCS 是一个支持 LU、Cholesky、Conjugate - gradient 等方法解稀疏线性方程的 C 语言函数库。第一版发布于 2001 年，但 TAUCS 已于 2003 年停止开发。

（6）BLAS（Basic Linear Algebra Subprograms）是一个线性代数的基本函数库，提供基本的向量和矩阵操作，Level - 1 的 BLAS 提供向量操作，Level - 2 的 BLAS 提供矩阵和向量之

间的操作，Level－3 的 BLAS 提供矩阵和矩阵之间的操作。

（7）TBB（Intel's Threading Building Blocks）是一个为各种算法提供多核，并行功能的 C ++ 函数库。

（8）LAPACK（Linear Algebra PACKage）是一个使用 Fortran 语言编写的，用来解决线性方程组、最小二次方、特征值、奇异值的函数库，并提供 LU、Cholesky、QR、SVD 等分解函数，不支持稀疏矩阵。

（9）AMD（Approximate Minimum Degree ordering）是一个对稀疏矩阵进行 Cholesky 和 LU 分解前做重新排列的函数库，有 C、Fortran 语言版本和 MATLAB 接口。

（10）CAMD（Constrained Approximate Minimum Degree ordering）也是一个对稀疏矩阵进行 Cholesky 和 LU 分解前做重新排列的函数库，有 C、Fortran 语言版本和 MATLAB 接口。

（11）COLAMD（COLumn Approximate Minimum Degree ordering）是另一个重新排列的函数库。

（12）CCOLAMD（Constrained COLumn Approximate Minimum Degree ordering）是另一个重新排列的函数库，和 COLAMD 类似。

（13）METIS 也是一个用于矩阵处理的函数库。

（14）ARPACK ++ 是一个用来解决大规模矩阵特征值的函数库。ARPACK ++ 在用 Fortran 语言编写的 ARPACK 上进行了 C ++ 语言的封装，提供了 C ++ 语言的接口。这个函数库可以用来计算矩阵的几个特征值和相应的特征向量。特别适合稀疏和有结构的矩阵。采用的算法是基于 Arnoldi 的 Implicit Restarted Arnoldi Method（IRAM）算法。加入矩阵是对称矩阵，那么采用的算法是基于 Lanczos 的 Implicitly Restarted Lanczos Method（IRLM）算法。对于很多操作，不需要进行矩阵的分解，而只需要对矩阵和向量进行操作。ARPACK 既可以处理单精度正定对称矩阵，也可以处理双精度复数非 Hermitian 矩阵。

（15）MOSEK 是一个用于解决优化问题的函数库。可以解决 Linear、Quadratic、Conic、Mixed Integer 等优化问题。提供 C、Java、C#、MATLAB 等多种语言接口。

（16）Intel Math Kernel Library 是一个在 Intel CPU 上优化的数学函数库。

（17）CUDA 是一个使用 NVIDIA GPU 做数学计算的函数库。

如图 17－1 所示列举了常用的线性系统数值计算函数库和相应的功能和数据结构。

Solver	分解方法	迭代方法	Matrix 种类	Out-of -core	存储结构
TAUCS	Colesky, LU,LDL^T	CG, Minres	All	Support	CCS
UMFPACK	LU	x	All	x	Triplet & CRS
CHOLMOD	Colesky	x	Symmetrical and Positive definite	x	Triplet & CRS
SuperLU	LU	x	All	x	CRS

图 17－1　线性系统数值计算函数库

17.1.2 软件下载

下面是上述的函数库下载地址，通过这些网址可以得到相关的函数库安装包。在下载完成后，进行编译和连接，就可以在C#程序中进行调用了。

（1）SuiteSparseQR：压缩包 SPQR. tar. gz

http://www. cise. ufl. edu/research/sparse/SPQR/current/

（2）CHOLMOD 压缩包 CHOLMOD. tar. gz

http://www. cise. ufl. edu/research/sparse/cholmod/current/

（3）UMFPACK 压缩包 UMFPACK. tar. gz

http://www. cise. ufl. edu/research/sparse/umfpack/current/

（4）SuperLU 压缩包 superlu_4. 3. tar. gz

http://crd - legacy. lbl. gov/ ~ xiaoye/SuperLU/

（5）TAUCS 压缩包 taucs_full. tgz

http://www. tau. ac. il/ ~ stoledo/taucs/

（6）GotoBlas

http://www. tacc. utexas. edu/tacc - projects/gotoblas2

（7）TBB

https://www. threadingbuildingblocks. org/download

（8）LAPACK 压缩包 lapack - 3. 4. 2. tgz

http://www. netlib. org/lapack/

（9）AMD：压缩包 AMD. tar. gz

http://www. cise. ufl. edu/research/sparse/amd/current/

（10）CAMD 压缩包 CAMD. tar. gz

http://www. cise. ufl. edu/research/sparse/camd/current/

（11）COLAMD 压缩包 COLAMD. tar. gz

http://www. cise. ufl. edu/research/sparse/colamd/current/

（12）CCOLAMD 压缩包，略

（13）METIS 压缩包 metis - 5. 1. 0. tar. gz

http://glaros. dtc. umn. edu/gkhome/metis/metis/download

（14）ARPACK ++ 压缩包 arpack ++. tar. gz

http://www.ime.unicamp.br/~chico/arpack++/
http://www.caam.rice.edu/software/ARPACK/

(15) MOSEK

http://mosek.com/introduction/try-mosek/

(16) Intel MKL

http://software.intel.com/en-us/intel-mkl

(17) CUDA

https://developer.nvidia.com/cuda-downloads

17.2　函数库编译

由于上述的大多数函数库都是在 UNIX 下编写的，所以在 Windows 下编译库时，需要使用 Cygwin 来模拟 UNIX 编译环境。即便编译完成能模拟 UNIX 下的使用，在 Windows 下 Visual Studio 单独使用 lib 文件也会不成功，所以在 Cygwin 下用 gcc 编译时，需要加上参数 -O3 -mno-cygwin。编译时经常会遇到 BLAS 库的情况，标准的 BLAS 库称 cBLAS，速度比较慢。除了标准 BLAS 以外，还有 GotoBLAS、Intel MKL、ATLAS 等 BLAS 库，速度比标准 BLAS 库快。

对稀疏矩阵来说，各种库都用类似相同的结构存储矩阵，用得比较多的是 CRS，即压缩行存储法。此外 Triplet 也用于存储矩阵，但 Triplet 占用空间会比 CRS 大，Triplet 和 CRS 的不同之处是处理行的索引，CRS 采用对列进行累加的形式，而 Triplet 则存储每一个矩阵项对应的列号。因此在大多数库中，先采用 Triplet 输入矩阵，再将其转换为 CRS 存储模式。额外的，TAUCS 采用的 CCS 与 CRS 类似，唯一不同是对行进行累计索引。

几种库之间的性能需要在解较大矩阵时才有稍微明显的差异，其中较快的是 TAUCS 和 CHOLMOD 采用的 Cholesky 分解法。LU 由于可分解非对称矩阵，对内存的消耗会较大。而 TAUCS 所采用的共轭梯度迭代解法，则需要提前设置迭代次数，迭代次数越多，结果越精确，最优迭代次数与矩阵数量正相关。对比行列数不同的矩阵，行数和列数较大的矩阵需要迭代更多次才能接近最终值。而对于较小矩阵，则不用设置较大迭代次数。

这几种库底层都采用了 BLAS 库，对于 BLAS 库的选择有 3 种：GotoBLAS、Intel MKL、ATLAS，这 3 种库都支持并行的 CPU 计算，其中 GotoBLAS 优化更好。对于 GPU 计算，CHOLMOD 在新版中加入了对 CUDA 的支持，因此在计算上采用 GPU 并行计算会更快捷。SuperLU 有并行计算的版本，支持多 PC 群中的内存共享，这种方式比 TAUCS 采用的 Out-of-core 效果会更好，当然也会更复杂。

编译完成的静态库需要转换为动态库的形式用于 C sharp 项目中，从简化代码的层面考虑，可以通过写 dll 接口使得 C#项目和解线性方程库进行交互，这样做可以减少 C#项目中直接调用底层库的复杂性，但其灵活性也降低。

测试编译完成的库是否可用也是一大问题，可以在 VS 建立一个 C++项目调用这些静

态库并遵照样例测试，如果成功再移植到动态库项目中。

1. UMFPACK 编译

1）编译静态链接库

下载源代码，同时 UMFPACK 依赖 SuiteSparse_config 包和 AMD 包，这两个包在以上网页中可找到下载链接。UMFPACK 是在 UNIX 环境下开发的，要在 Windows 下编译 UMFPACK，需要 Cygwin 来模拟 UNIX 编译环境。依照如下步骤在 Windows 下编译 UMFPACK。

第一步：下载并解压 UMFPACK、AMD 和 SuiteSparse_config 包到相同路径下。

第二步：BLAS 是可选的，BLAS 包在 UMFPACK 下主要用来加速计算。使用 BLAS 需要将 libatlas. a libblas. a libcblas. a libf77blas. a liblapack. a 添加到 UMFPACK 下的/lib 中。

第三步：转到路径/SuiteSpase_config 并修改 SuiteSparse_config. mk 中以下行内容：

```
CFLAGS = -O3 -mno-cygwin
BLAS = -L$(Path of BLAS libs) -lf77blas -latlas -lg2c
```

其中，第一行标签 -mno-cygwin 可使编译好的静态库文件独立于 Cygwin 环境运行。若编译 UMFPACK 需 BLAS，则按照第二行 BLAS 后面的参数进行设置。其中 -L$后的括号中是静态库的路径。

第四步：打开 Cygwin 的控制台窗口，进入 UMFPACK 的目录，使用 make 命令进行编译，若要重新编译或者清除编译好的库，使用命令 make purge。

所有编译好的 UMFPACK 库文件都在 /lib 文件夹下，将所有库文件重命名为 .lib。

2）创建动态链接库

制作动态链接库需要以下 UMFPACK 静态库。

```
libatlas.lib,libf77blas.lib,libg2c.lib,libgcc.lib,libumfpack.lib,libamd.lib,libsuitesparseconfig.lib
```

将以上静态库文件名添加到 Visual Studio 项目的“Project Properties”窗口中“Additional Dependencies”栏内，如图 17-2 所示。

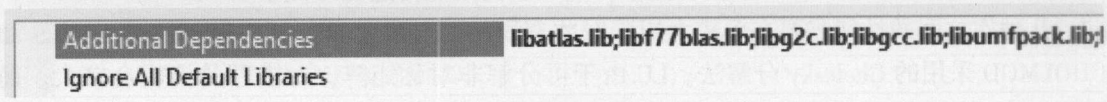


图 17-2　添加静态库文件名

设置附加 Include 和 Library 路径，将路径指向静态库文件的目录如图 17-3 所示。

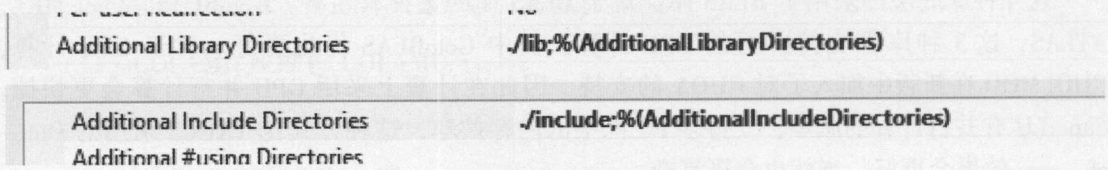


图 17-3　将路径指向静态库文件的目录下

最后一步设置 dll 文件生成路径，编译好的动态链接库将放在预先设置的目录如图 17-4 所示。

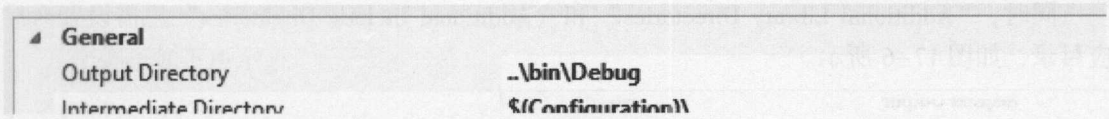


图 17-4　预先设备的目录

在项目对应的头文件中，所有需要被外部调用的函数都在其声明前加上下例标签：

```
extern "C" __declspec( dllexport)
```

2. Cholmod 编译

1）编译静态链接库

CHOLMOD 同时需要 SuiteSparse_config、AMD、COLAMD、CAMD、CCOLAMD 等包，METIS、Lapack、BLAS 和 Goto BLAS 是可选的包。

编译的步骤与 UMFPACK 相似。将 Metis、Lapack、BLAS 的库文件复制到/lib 路径下，并在 Cygwin 环境下使用命令 make 编译，并且编辑以下标签：

```
CFLAGS = -O3 -mno-cygwin
BLAS = -L$( Path of BLAS libs ) -lf77blas -latlas -lg2c
```

2）创建动态链接库

与 UMFPACK 除了静态库配置外，其他与 UMFPACK 相同。替换“Additional Dependencies”中的静态库文件列表。

```
libatlas.lib, libf77blas.lib, libg2c.lib, libgcc.lib, liblapack.lib, libcamd.lib, libccolamd.lib, metis.lib,
libtmglib.lib, libcholmod.lib, libcolamd.lib, libamd.lib, libsuitesparseconfig.lib, libgfortran.lib, lib-
mingw32.lib, libmingwex.lib, liblapacke.lib
```

3. TAUCS 编译

1）编译静态链接库

TAUCS 可以在 Windows 下单独编译，运行 configure.bat 批处理文件来更新系统信息。在 Console 窗口，转到 TAUCS 目录并使用命令 nmake，编译完成后，静态库文件 libtaucs.lib 可以在 TAUCS 下的 /lib 目录下找到。

2）创建动态库

要使用 TAUCS 库，需要下列外部静态库作为依赖：

```
libtaucs.lib, blas_win32.lib, libatlas.lib, libcblas.lib, libmetis.lib, libf77blas.lib,
liblapack.lib, vcf2c.lib
```

注意：静态库文件 libcmt.lib 会在之后的编译中造成错误，所以需要将其在设置中排除。最终设置如图 17-5 所示。

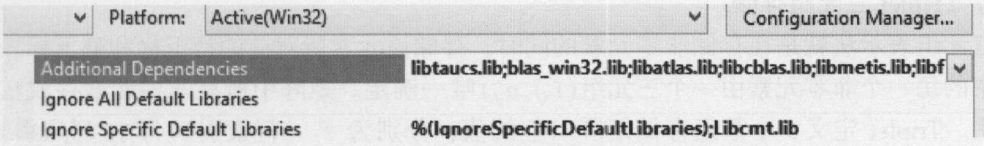


图 17-5　最终设置

同时，“Additional Library Directories”和“Additional Include Directories”栏需设置为对应目录，如图 17-6 所示。

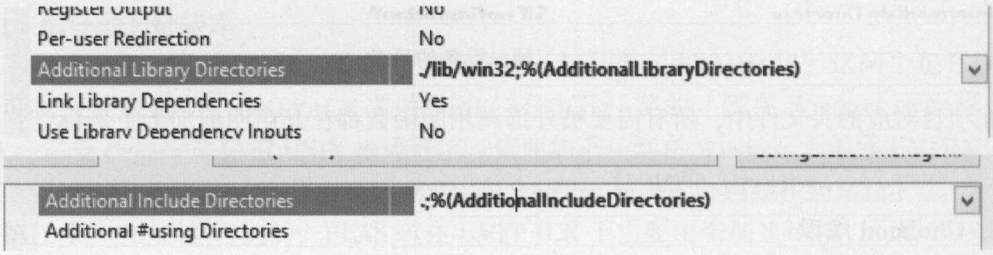


图 17-6　设置为对应目录

4. SuperLU 编译

由于 SuperLU 需要 BLAS 库支持，所以编译 SuperLU 前需要编译好的 BLAS 库，基本的 BLAS 库较慢，因此推荐 Intel MKL、GotoBLAS、ATLAS。这里以 ATLAS 为例。

编译 ATLAS 需要 LAPACK 支持，在 Windows 下需要 Cygwin 支持，方法可参考编译 UMFPACK。将编译好的 ATLAS 放入 SuperLU 的/lib 文件夹下。修改 SuperLU 目录下的 make. ini 文件，完成以下步骤。

第一步：将 SuperLUroot 修改为 SuperLU 的绝对路径。

第二步：修改 BLAS 路径为 Atlas 库路径，并在后面加上库名，如：

```
BLASLIB = -L$(SuperLUroot)/lib/ -lblas
```

第三步：修改 CFLAGS 为 -O3 -mno-cygwin，-mno-cygwin 可以使编译后的 lib 文件在 Visual Studio 项目中独立使用。

第四步：在 Cygwin 控制台下用 make 命令进行编译，生成的静态库文件在 SuperLU 根目录下。

17.3　稀疏矩阵

在线性系统的计算中，为了加快计算和减少线性系统的内存占有量，需要采用高效的存储结构来存储矩阵。如果在矩阵中，多数的元素为零，称此矩阵为稀疏矩阵（Sparse Matrix）。稀疏矩阵中非零元素的个数远远小于矩阵元素的总数，并且非零元素的分布没有规律。与之相区别的是，如果非零元素的分布存在规律（如上三角矩阵、下三角矩阵、对称矩阵），则称该矩阵为特殊矩阵。对于稀疏矩阵来说，若采用二维数组来表示矩阵，在矩阵较大时，会产生较大的空间浪费。零值的项目并不需要存储，采用压缩存储方法可以有效减少空间消耗。稀疏矩阵的存储有三种类型：CCS、CRS 和 Triplet。

1. Triplet 三元组存储

三元组表示法就是在存储非零元素的同时，存储该元素所对应的行下标和列下标。稀疏矩阵中的每一个非零元素由一个三元组(i,j,a_{ij})唯一确定。矩阵中所有非零元素存放在 3 个数组中。Triplet 定义 3 个数组存储矩阵对应的值，分别为 T_i（行索引）、T_j（列索引）、T_x（值）。T_x的值与 T_i、T_j 的值相关，同时 Triplet 中也需要记录矩阵中非零项的总数 nnz 。对

于整数 $i(i<nnz)$，行 $T_i[i]$ 与列 $T_j[i]$ 的值为 $T_x[i]$。

几个例子如下所示。

$$\begin{bmatrix}3 & 0 & 0\\0 & 1 & 1\\1 & 2 & 0\end{bmatrix}$$

$$T_x=\{3,1,1,2,1\};$$

$$T_i=\{0,2,1,2,1\};$$

$$T_j=\{0,0,1,1,2\}$$

$$A_{6\times7}=\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0\\0 & 2 & 0 & 0 & 0 & 0 & 0\\3 & 0 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 0 & 5 & 0 & 0 & 0\\0 & 0 & 0 & 0 & 6 & 0 & 0\\0 & 0 & 0 & 0 & 0 & 7 & 4\end{bmatrix}$$

序号	行	列	值
0	6	7	7
1	0	2	1
2	1	1	2
3	2	0	3
4	3	3	5
5	4	4	6
6	5	5	7
7	5	6	4

2. CRS（Compressed Row Storage）按行压缩存储

CRS 数据结构使用 3 个数组来表示稀疏矩阵，第 1 个数组按行存放所有非零值，第 2 个数组存放非零值所在的列，第 3 个数组存放每行的第 1 个非零值在第 1 个数组里面的序号。CRS 与 Triplet 不同的地方是将列压缩为累计索引，下面通过例子说明。

$$\begin{bmatrix}3 & 0 & 0\\0 & 1 & 1\\1 & 2 & 0\end{bmatrix}$$

$$A_x=\{3,1,1,2,1\};$$

$$A_i=\{0,2,1,2,1\};$$

$$A_p=\{0,2,4,5\};$$

A_x、A_i 与 Triplet 中 T_x、T_i 相同，唯一不同的是数组 A_p。A_p 为列项累加计数索引，数组个数为矩阵列数加 1。如上矩阵，第 1 列有 2 项目，第 2 列有 2 项目，第 3 列有 1 项目。A_p 的第 2 项为第 1 列的累加计数，A_p 的第 3 项为第 1 列计数加第 2 列计数所有的总和，以此类推，最后一项为 5，与 Triplet 的非零总数相等。

3. CCS（Compressed Column Storage）按列压缩存储

CCS（Compressed Column Storage）是指按列压缩存储，CCS 和 CRS 类似，只是按照列来存放非零值，是 CRS 的转置。

17.4　函数库调用

1. 函数声明

在 C#语言中调用 C ++ 语言封装的动态链接库，需要在文件里声明相关的函数库，还要

把相关的函数库放到可执行目录下。

（1）例如，如果要使用 SuperLU 动态链接库中的 3 个函数，声明这 3 个函数所在的函数库的方式如下。其他函数库的声明类似。

```
[DllImport("SuperLU.dll", CharSet = CharSet.Unicode,
    CallingConvention = CallingConvention.Cdecl)]
public static extern unsafe void* CreateSolverLUSuperLU(int numberOfRows,
    int numberOfColumns, int numberOfNoneZero, int* rowIndex,
    int* columnIndex, double* values);

[DllImport("SuperLU.dll", CharSet = CharSet.Unicode,
    CallingConvention = CallingConvention.Cdecl)]
public static extern unsafe void SolveLUSuperLU(void* solver,
    double* x, double* b);

[DllImport("SuperLU.dll", CharSet = CharSet.Unicode,
    CallingConvention = CallingConvention.Cdecl)]
public static extern unsafe void FreeSolverLUSuperLU(void* solver);
```

（2）调用 Arpack 函数库计算特征值和特征向量的动态链接库声明如下。

```
[DllImport("EigenArpackUtil.dll", CharSet = CharSet.Unicode,
    CallingConvention = CallingConvention.Cdecl)]
public static extern unsafe
int ComputeEigenNoSymmetricShiftModeCRS(
    int* index,
    int* pointer,
    double* values,
    int numberOfNoneZeros,
    int numberOfRows,
    int resultCount,
    double sigma,
    double* RealPart,
    double* ImagePart,
    double* EigenVectors
    );
```

2. 函数使用

在动态链接库里封装的函数声明之后，就可以和普通 C#函数一样调用这些函数了。

（1）求解特征值和特征向量。

```
public Eigen ComputeEigensByLib(SparseMatrixDouble sparse,
    double sigma, int count)
{
```

```
    int[ ] pCol;
    int[ ] iRow;
    double[ ] Values;
    int NNZ;
    int m = sparse. RowCount;
    sparse. ToCCS(out pCol, out iRow, out Values, out NNZ);
    double[ ] ImagePart = new double[count];
    double[ ] RealPart = new double[count];
    double[ ] Vectors = new double[count * m];
    fixed (int* ri = iRow, cp = pCol)
    fixed (double* val = Values, vets = Vectors,
        imgPart = ImagePart, relPart = RealPart)
    {
        int result =
            ComputeEigenNoSymmetricShiftModeCRS(ri, cp,
            val, NNZ, m,count, sigma, relPart, imgPart, vets);
    }
    List<EigenPair> list = new List<EigenPair>();
    for (int i = 0; i < count; i++)
    {
        double realPart = RealPart[i];
        List<double> vector = new List<double>();
        int startIndex = i * m;
        int endIndex = i * m + m;
        for (int j = startIndex; j < endIndex; j++)
        {
            double value = Vectors[j];
            vector. Add(value);
        }

        EigenPair newPair =new EigenPair(realPart, vector);
        list. Add(newPair);
    }
    list. Sort();
    Eigen eigen =new Eigen();
    eigen. SortedEigens = list. ToArray();
    return eigen;
}
```

(2) 稀疏矩阵分解。

```
public void Factorization(SparseMatrix A)
{
```

```
        TripletArraryData data = ConvertToTripletArrayData(A);

        int rowCount = A.Rows.Count;
        int columnCount = A.Columns.Count;
        int nnz = data.nnz;

        fixed (int* ri = data.rowIndex, ci = data.colIndex)
        fixed (double* val = data.values)
        {
            switch (SolverType)
            {

                case EnumSolver.UmfpackLU:
                    solver = CreateSolverLUUMFPACK(rowCount, nnz,
                        ri, ci, val);
                    break;
                case EnumSolver.SuperLULU:
                    solver = CreateSolverLUSuperLU(rowCount, rowCount,
                        nnz, ri, ci, val);
                    break;
                case EnumSolver.CholmodCholesky:
                    solver = CreateSolverCholeskyCHOLMOD(rowCount, rowCount,
                        nnz, nnz, ri, ci, val);
                    break;
                case EnumSolver.SPQRLeastNormal:
                    solver = CreateSolverQRSuiteSparseQR(rowCount, columnCount,
                        nnz, nnz, ri, ci, val);
                    break;
                case EnumSolver.SPQRLeastSqure:
                    solver = CreateSolverQRSuiteSparseQR(rowCount, columnCount,
                        nnz, nnz, ri, ci, val);
                    break;
            }
        }
        if (solver == null) throw new Exception("Create Solver Fail");
    }
```

（3）求解线性方程组。

由于有多种函数库都可以求解线性系统，因此可以用如下代码来选择使用哪一个函数库。

```
    public void SolveLinerSystem(ref double[] rightB, ref double[] unknownX)
    {
```

```
        if (solver == null)
            return;
        fixed (double * _x = unknownX, _rightB = rightB)
        {
            switch (SolverType)
            {
                case EnumSolver.TaucsCholesky:
                    SolveCholeskyTAUCS(solver, _x, _rightB);
                    break;
                case EnumSolver.TaucsCG:
                    SolveCGTAUCS(solver, _x, _rightB);
                    break;
                case EnumSolver.TaucsLU:
                    SolveLUTAUCS(solver, _x, _rightB);
                    break;
                case EnumSolver.UmfpackLU:
                    SolveLUUMFPACK(solver, _x, _rightB);
                    break;
                case EnumSolver.SuperLULU:
                    SolveLUSuperLU(solver, _x, _rightB);
                    break;
                case EnumSolver.CholmodCholesky:
                    SolveCholeskyCHOLMOD(solver, _x, _rightB);
                    break;
                case EnumSolver.SPQRLeastNormal:
                    SolveLeastNormalByQR(solver, _x, _rightB);
                    break;
                case EnumSolver.SPQRLeastSqure:
                    SolveLeastSqureByQR(solver, _x, _rightB);
                    break;
            }
        }
    }
```

(4) 求解线性系统。

线性系统根据矩阵的行数和列数相等、行数大于列数、行数小于列数可以分为方阵、Least - Square、Least - Normal 三种方式进行求解，代码如下所示。

```
    public double[] SloveLeastSquare(ref SparseMatrix L, ref double[] b)
    {
        double[] result = new double[L.ColumnSize];

        LinearSystemLib linearSolver = new LinearSystemLib();
```

```
        linearSolver. SolverType = EnumSolver. SPQRLeastSqure;
        linearSolver. Factorization( L) ;
        linearSolver. SolveLinerSystem( ref b, ref result) ;
        linearSolver. FreeSolver( ) ;
        return result;
    }

    public double[ ] SloveLeastNormal( ref SparseMatrix L, ref double[ ] b)
    {
        double[ ] result = new double[ L. ColumnSize] ;
        LinearSystemLib linearSolver = new LinearSystemLib( ) ;
        linearSolver. SolverType = EnumSolver. SPQRLeastNormal;
        linearSolver. Factorization( L. Transpose( ) ) ;
        linearSolver. SolveLinerSystem( ref b,ref result) ;
        linearSolver. FreeSolver( ) ;
        return result;
    }

    public double[ ] SloveSquare( ref SparseMatrix A, ref double[ ] rightB)
    {
        double[ ] unknown = new double[ rightB. Length] ;
        LinearSystemLib linearSolver = new LinearSystemLib( EnumSolver. UmfpackLU) ;
        linearSolver. Factorization( A) ;
        linearSolver. SolveLinerSystem( ref rightB, ref unknown) ;
        linearSolver. FreeSolver( ) ;
        return unknown;
    }
```

参考文献

[1] Sander, Pedro V., et al. Texture mapping progressive meshes. Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001.

[2] Sorkine, Olga, et al. Bounded - distortion piecewise mesh parameterization. Proceedings of the conference on Visualization02. IEEE Computer Society, 2002.

[3] Lévy, Bruno, et al. Least squares conformal maps for automatic texture atlas generation. ACM Transactions on Graphics (TOG). Vol. 21. No. 3. ACM, 2002.

[4] Liu, Ligang, et al. A local/global approach to mesh parameterization. Computer Graphics Forum. Vol. 27. No. 5. Blackwell Publishing Ltd, 2008.

[5] Kharevych, Liliya, Boris Springborn, and Peter Schröder. Discrete conformal mappings via circle patterns. ACM Transactions on Graphics (TOG) 25.2 (2006): 412 - 438.

[6] Ben - Chen, Mirela, Craig Gotsman, and Guy Bunin. Conformal flattening by curvature prescription and metric scaling. Computer Graphics Forum. Vol. 27. No. 2. Blackwell Publishing Ltd, 2008.

[7] Kälberer, Felix, Matthias Nieser, and Konrad Polthier. QuadCover - Surface Parameterization using Branched Coverings. Computer Graphics Forum. Vol. 26. No. 3. Blackwell Publishing Ltd, 2007.

[8] Sheffer, Alla, Emil Praun, and Kenneth Rose. Mesh parameterization methods and their applications. Foundations and Trends® in Computer Graphics and Vision 2.2 (2006): 105 - 171.

[9] Maillot, Jérôme, Hussein Yahia, and Anne Verroust. Interactive texture mapping. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. ACM, 1993.

[10] Lee, Yunjin, Hyoung Seok Kim, and Seungyong Lee. Mesh parameterization with a virtual boundary. Computers & Graphics 26.5 (2002): 677 - 686.

[11] Zayer, Rhaleb, Christian Rössl, and Hans - Peter Seidel. Setting the Boundary Free: A Composite Approach to Surface Parameterization. Symposium on Geometry Processing. 2005.

[12] Desbrun, Mathieu, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. Computer Graphics Forum. Vol. 21. No. 3. Blackwell Publishing, Inc, 2002.

[13] Gu, Xianfeng, and Shing - Tung Yau. Global conformal surface parameterization. Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing. Eurographics Association, 2003.

[14] Hormann K., Polthier K., Sheffer A.: SIGGRAPH asia 2008 course notes mesh parameterization: theory and practice.

[15] Myles, Ashish, and Denis Zorin. Global parametrization by incremental flattening. ACM Transactions on Graphics (TOG) 31.4 (2012): 109.

[16] Chen, Zhonggui, et al. Surface parameterization via aligning optimal local flattening. Proceedings of the 2007 ACM symposium on Solid and physical modeling. ACM, 2007.

[17] Sheffer, Alla, et al. ABF++: fast and robust angle based flattening. ACM Transactions on Graphics (TOG)

24.2（2005）：311－330.

[18] Springborn, Boris, Peter Schröder, and Ulrich Pinkall. Conformal equivalence of triangle meshes. ACM Transactions on Graphics（TOG）27.3（2008）：77.

[19] Jin, Miao, et al. Conformal surface parameterization using euclidean ricci flow. Technique report, May（2006）.

[20] Gortler, Steven J., Craig Gotsman, and Dylan Thurston. Discrete one－forms on meshes and applications to 3D mesh parameterization. Computer Aided Geometric Design 23.2（2006）：83－112.

[21] Hormann, Kai, and Günther Greiner. MIPS：An efficient global parametrization method. ERLANGEN－NUERNBERG UNIV（GERMANY）COMPUTER GRAPHICS GROUP, 2000.

▶ 作者邮箱：graphicsresearch@qq.com

VR 三维技术系列

VR三维技术系列图书简介

着重底层核心技术的讲解，涵盖三维图形学算法的三大方面，即建模、动画和渲染。读者通过学习本系列专业基础图书和现有成熟计算机基础编程教材，以及三维软件使用的图书，知识体系可以完整地覆盖数字媒体技术专业的所有课程。

书中代码采用C#编程语言，但是本套系列图书里讲述的原理和算法不仅限于C#语言。读者可以通过示例中的代码，采用自己熟悉的编程语言进行编程。本套系列图书包含了很多计算机图形学会议Siggraph论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现与分析。

读者对象

虚拟现实从业人员、数字媒体专业师生、三维游戏工程师、计算机专业师生、软件工程专业师生、影视特效工程师等。

电子信息出版分社微博
http://weibo.com/etpublish

策划编辑：张　迪
责任编辑：底　波
封面设计：徐海燕

官方微信平台

ISBN 978-7-121-31682-1

定价：79.00 元